



TATA McGRAW-HILL EDITION  
FOR SALE IN INDIA ONLY

International Best-Seller!

Over 200,000 Copies Sold!

# TEACH YOURSELF

"Herb Schildt tells his programmers what they want and need to know—simply, clearly, concisely, and authoritatively."

—ACM Computing Reviews



Third  
Edition

The Most Successful and Proven Method for Learning C++

- Master Standard C++ – Including its new features! •
- Essential for C Programmers moving on to C++ •
- Build your skills with hundreds of examples and exercises •
- Covers the Standard Template Library (STL) •



# Herbert Schildt

Best-Selling C/C++ Author with More Than 1.5 Million Books Sold

Copyrighted material



**Tata McGraw-Hill**

## **Teach Yourself C++, Third Edition**

Copyright © 1998 by The McGraw-Hill Companies , Inc.,  
All rights reserved.

No part of this publication may be reproduced or distributed in any  
form or by any means, or stored in a data base or retrieval system,  
without the prior written permission of the publisher, with the exception  
that the program listings may be entered, stored, and executed in a computer  
system, but they may not be reproduced for publication

### **Tata McGraw-Hill Edition 1998**

**Twentieth reprint 2008**  
**RZZLCDRXRBYDY**

**Reprinted in India by arrangement with The McGraw-Hill Companies, Inc.,  
New York**

### **For Sale in India Only**

**ISBN-13: 978-0-07-043870-5**  
**ISBN-10: 0-07-043870-X**

**Published by Tata McGraw-Hill Publishing Company Limited,  
7 West Patel Nagar, New Delhi 110 008, and printed at  
Krishna Offset, Delhi 110 032**

**The McGraw-Hill Companies**

## Contents at a Glance

<u>1</u>	<u>An Overview of C++</u>	<u>1</u>
2	<i>Introducing Classes</i>	41
3	<i>A Closer Look at Classes</i>	87
4	<i>Arrays, Pointers, and References</i>	117
5	<i>Function Overloading</i>	159
6	<i>Introducing Operator Overloading</i>	195
7	<i>Inheritance</i>	231
8	<i>Introducing the C++ I/O System</i>	269
<u>9</u>	<u>Advanced C++ I/O</u>	<u>307</u>
<u>10</u>	<u>Virtual Functions</u>	<u>345</u>
11	<i>Templates and Exception Handling</i>	371
12	<i>Run-Time Type Identification and the Casting Operators</i>	407
13	<i>Namespaces, Conversion Functions, and Miscellaneous Topics</i>	435
14	<i>Introducing the Standard Template Library</i>	473
A	<i>A Few More Differences Between C and C++</i>	531
B	<i>Answers</i>	533
	<u>Index</u>	<u>739</u>

---

## **A**bout the Author...

Herbert Schildt is the world's leading programming author. He is an authority on the C and C++ languages, a master Windows programmer, and an expert on Java. His programming books have sold nearly two million copies worldwide and have been translated into all major foreign languages. He is the author of numerous best-sellers, including *C: The Complete Reference*, *C++: The Complete Reference*, *C++ from the Ground Up*, *Expert C++*, *MFC Programming from the Ground Up*, *Windows 95 Programming in C and C++*, *Windows NT 4 Programming from the Ground Up*, and many others. Schildt is the president of Universal Computing Laboratories, a software consulting firm in Mahomet, Illinois. He is also a member of both the ANSI C and C++ standardization committees. He holds a master's degree in computer science from the University of Illinois.

# Contents

Acknowledgments, xiii

Introduction, xv

For Further Study, xix

## **1 An Overview of C++ . . . 1**

- 1.1 WHAT IS OBJECT-ORIENTED PROGRAMMING?, 3**
- 1.2 TWO VERSIONS OF C++, 7**
- 1.3 C++ CONSOLE I/O, 13**
- 1.4 C++ COMMENTS, 19**
- 1.5 CLASSES: A FIRST LOOK, 21**
- 1.6 SOME DIFFERENCES BETWEEN C AND C++, 28**
- 1.7 INTRODUCING FUNCTION OVERLOADING, 33
- 1.8 C++ KEYWORDS, 39
- SKILLS CHECK, 39

## **2 Introducing Classes . . . 41**

- 2.1 CONSTRUCTOR AND DESTRUCTOR FUNCTIONS, 43
- 2.2 CONSTRUCTORS THAT TAKE PARAMETERS, 52
- 2.3 INTRODUCING INHERITANCE, 59
- 2.4 OBJECT POINTERS, 66
- 2.5 CLASSES, STRUCTURES, AND UNIONS ARE RELATED, 68
- 2.6 IN-LINE FUNCTIONS, 75
- 2.7 AUTOMATIC IN-LINING, 80
- SKILLS CHECK, 83

## **3 A Closer Look at Classes . . . 87**

- 3.1 ASSIGNING OBJECTS, 89
- 3.2 PASSING OBJECTS TO FUNCTIONS, 96

3.3	RETURNING OBJECTS FROM FUNCTIONS,	102
3.4	AN INTRODUCTION TO FRIEND FUNCTIONS,	107
	SKILLS CHECK,	114

#### **4 Arrays, Pointers, and References . . . 117**

4.1	ARRAYS OF OBJECTS,	119
4.2	USING POINTERS TO OBJECTS,	124
4.3	THE this POINTER,	126
4.4	USING new AND delete,	130
4.5	MORE ABOUT new AND delete,	134
<b>4.6</b>	<b>REFERENCES,</b>	<b>140</b>
4.7	PASSING REFERENCES TO OBJECTS,	146
4.8	RETURNING REFERENCES,	149
4.9	INDEPENDENT REFERENCES AND RESTRICTIONS,	154
	SKILLS CHECK,	156

#### **5 Function Overloading . . . 159**

5.1	OVERLOADING CONSTRUCTOR FUNCTIONS,	161
5.2	CREATING AND USING A COPY CONSTRUCTOR,	167
5.3	THE overload ANACHRONISM,	177
5.4	USING DEFAULT ARGUMENTS,	177
<b>5.5</b>	<b>OVERLOADING AND AMBIGUITY,</b>	<b>185</b>
5.6	FINDING THE ADDRESS OF AN OVERLOADED FUNCTION,	189
	<b>SKILLS CHECK,</b>	<b>191</b>

#### **6 Introducing Operator Overloading . . . 195**

6.1	THE BASICS OF OPERATOR OVERLOADING,	197
6.2	OVERLOADING BINARY OPERATORS,	199
6.3	OVERLOADING THE RELATIONAL AND LOGICAL OPERATORS,	207
6.4	OVERLOADING A UNARY OPERATOR,	209
6.5	USING FRIEND OPERATOR FUNCTIONS,	213
6.6	A CLOSER LOOK AT THE ASSIGNMENT OPERATOR,	218
6.7	OVERLOADING THE [ ] SUBSCRIPT OPERATOR,	222
	SKILLS CHECK,	227

<b>7 Inheritance . . .</b>	<b>231</b>
7.1	BASE CLASS ACCESS CONTROL, 234
7.2	USING PROTECTED MEMBERS, 240
7.3	CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE, 244
7.4	MULTIPLE INHERITANCE, 252
7.5	VIRTUAL BASE CLASSES, 259
	SKILLS CHECK, 262

## **8 Introducing the C++ I/O System . . .** **269**

8.1	SOME C++ I/O BASICS, 273
8.2	FORMATTED I/O, 275
8.3	USING width( ), precision( ), AND fill( ), 283
8.4	USING I/O MANIPULATORS, 287
8.5	CREATING YOUR OWN INSERTERS, 292
8.6	CREATING EXTRACTORS, 299
	SKILLS CHECK, 303

## **9 Advanced C++ I/O . . .** **307**

9.1	<u>CREATING YOUR OWN MANIPULATORS,</u> 309
9.2	FILE I/O BASICS, 313
9.3	UNFORMATTED, BINARY I/O, 320
9.4	MORE UNFORMATTED I/O FUNCTIONS, 327
9.5	RANDOM ACCESS, 331
9.6	CHECKING THE I/O STATUS, 334
9.7	CUSTOMIZED I/O AND FILES, 338
	SKILLS CHECK, 341

## **10 Virtual Functions . . .** **345**

10.1	<u>POINTERS TO DERIVED CLASSES,</u> 347
10.2	INTRODUCTION TO VIRTUAL FUNCTIONS, 349
10.3	MORE ABOUT VIRTUAL FUNCTIONS, 357
10.4	APPLYING POLYMORPHISM, 362
	SKILLS CHECK, 368

## **11 Templates and Exception Handling . . .** **371**

11.1	GENERIC FUNCTIONS, 373
11.2	GENERIC CLASSES, 380
11.3	EXCEPTION HANDLING 386
11.4	MORE ABOUT EXCEPTION HANDLING, 394

11.5 HANDLING EXCEPTIONS THROWN  
BY new, 401  
SKILLS CHECK, 404

## **12 Run-Time Type Identification and the Casting Operators . . . 407**

12.1 UNDERSTANDING RUN-TIME TYPE  
IDENTIFICATION (RTTI), 409  
12.2 USING dynamic\_cast, 420  
12.3 USING const\_cast, reinterpret\_cast, AND  
static\_cast, 429  
SKILLS CHECK, 432

## **13 Namespaces, Conversion Functions, and Miscellaneous Topics . . . 435**

13.1 NAMESPACES, 437  
13.2 CREATING A CONVERSION FUNCTION, 446  
13.3 STATIC CLASS MEMBERS, 449  
13.4 const MEMBER FUNCTIONS AND mutable, 455  
13.5 A FINAL LOOK AT CONSTRUCTORS, 459  
13.6 USING LINKAGE SPECIFIERS AND THE asm  
KEYWORD, 463  
13.7 ARRAY-BASED I/O, 466  
SKILLS CHECK, 471

## **14 Introducing the Standard Template Library . . . 473**

14.1 AN OVERVIEW OF THE STANDARD TEMPLATE  
LIBRARY, 476  
14.2 THE CONTAINER CLASSES, 479  
14.3 VECTORS, 480  
14.4 LISTS, 490  
14.5 MAPS, 502  
14.6 ALGORITHMS, 509  
14.7 THE string CLASS, 519  
SKILLS CHECK, 529

## **A A Few More Differences Between C and C++ . . . 531**

## **B Answers . . . 533**

1.3 EXERCISES, 534  
1.4 EXERCISES, 535

<u>1.5</u>	<u>EXERCISES,</u>	535
<u>1.6</u>	<u>EXERCISES,</u>	538
<u>1.7</u>	<u>EXERCISES,</u>	538
	MASTER SKILLS CHECK: Chapter 1,	541
	REVIEW SKILLS CHECK: Chapter 2,	543
<u>2.1</u>	<u>EXERCISES,</u>	545
<u>2.2</u>	<u>EXERCISES,</u>	548
<u>2.3</u>	<u>EXERCISE,</u>	551
<u>2.5</u>	<u>EXERCISES,</u>	553
<u>2.6</u>	<u>EXERCISES,</u>	555
<u>2.7</u>	<u>EXERCISES,</u>	556
	MASTER SKILLS CHECK: Chapter 2,	558
	CUMULATIVE SKILLS CHECK: Chapter 2,	560
	REVIEW SKILLS CHECK: Chapter 3,	562
<u>3.1</u>	<u>EXERCISES,</u>	563
<u>3.2</u>	<u>EXERCISES,</u>	565
<u>3.3</u>	<u>EXERCISES,</u>	567
<u>3.4</u>	<u>EXERCISE,</u>	567
	MASTER SKILLS CHECK: Chapter 3,	569
	CUMULATIVE SKILLS CHECK: Chapter 3,	571
	<u>REVIEW SKILLS CHECK: Chapter 4,</u>	<u>576</u>
<u>4.1</u>	<u>EXERCISES,</u>	578
<u>4.2</u>	<u>EXERCISES,</u>	580
<u>4.3</u>	<u>EXERCISE,</u>	582
<u>4.4</u>	<u>EXERCISES,</u>	583
<u>4.5</u>	<u>EXERCISES,</u>	584
<u>4.6</u>	<u>EXERCISES,</u>	585
<u>4.7</u>	<u>EXERCISE,</u>	586
<u>4.8</u>	<u>EXERCISES,</u>	587
	<u>MASTER SKILLS CHECK: CHAPTER 4,</u>	<u>589</u>
	CUMULATIVE SKILLS CHECK: Chapter 4,	592
	REVIEW SKILLS CHECK: Chapter 5,	593
<u>5.1</u>	<u>EXERCISES,</u>	595
<u>5.2</u>	<u>EXERCISES,</u>	598
<u>5.4</u>	<u>EXERCISES,</u>	600
<u>5.6</u>	<u>EXERCISE,</u>	601
	MASTER SKILLS CHECK: Chapter 5,	602
	<u>CUMULATIVE SKILLS CHECK: Chapter 5,</u>	<u>605</u>
	REVIEW SKILLS CHECK: Chapter 6,	607

<u>6.2</u>	<u>EXERCISES,</u>	608
6.3	EXERCISE,	609
<u>6.4</u>	<u>EXERCISES,</u>	610
<u>6.5</u>	<u>EXERCISES,</u>	612
<u>6.6</u>	<u>EXERCISE,</u>	616
6.7	EXERCISES,	618
	<u>MASTERY SKILLS CHECK: Chapter 6,</u>	621
	CUMULATIVE SKILLS CHECK: Chapter 6,	629
	<u>REVIEW SKILLS CHECK: Chapter 7,</u>	631
7.1	EXERCISES,	637
7.2	EXERCISES,	637
<u>7.3</u>	<u>EXERCISES,</u>	638
<u>7.4</u>	<u>EXERCISES,</u>	640
<u>7.5</u>	<u>EXERCISES,</u>	641
	<u>MASTERY SKILLS CHECK: Chapter 7,</u>	641
	CUMULATIVE SKILLS CHECK: Chapter 7,	643
	<u>REVIEW SKILLS CHECK: Chapter 8,</u>	644
8.2	EXERCISES,	646
<u>8.3</u>	<u>EXERCISES,</u>	647
8.4	EXERCISES,	649
8.5	EXERCISES,	650
8.6	EXERCISES,	652
	<u>MASTERY SKILLS CHECK: Chapter 8,</u>	655
	CUMULATIVE SKILLS CHECK: Chapter 8	659
	<u>REVIEW SKILLS CHECK: Chapter 9,</u>	662
9.1	EXERCISES,	664
9.2	EXERCISES,	666
9.3	EXERCISES,	668
<u>9.4</u>	<u>EXERCISES,</u>	671
9.5	EXERCISES,	673
9.6	EXERCISE,	674
	<u>MASTERY SKILLS CHECK: Chapter 9,</u>	677
	CUMULATIVE SKILLS CHECK: Chapter 9,	682
	<u>REVIEW SKILLS CHECK: Chapter 10,</u>	684
10.2	EXERCISES,	687
10.3	EXERCISES,	688
10.4	EXERCISE,	689
	<u>MASTERY SKILLS CHECK: Chapter 10,</u>	694
	CUMULATIVE SKILLS CHECK: Chapter 10,	694
	<u>REVIEW SKILLS CHECK: Chapter 11,</u>	698

<u>11.1</u>	EXERCISES,	698
<u>11.2</u>	EXERCISES,	699
<u>11.3</u>	EXERCISES,	701
11.4	EXERCISES,	702
11.5	EXERCISES,	702
	<u>MASTERY SKILLS CHECK: Chapter 11,</u>	703
	REVIEW SKILLS CHECK: Chapter 12,	710
<u>12.1</u>	EXERCISES,	712
<u>12.2</u>	EXERCISES,	712
12.3	EXERCISES,	713
	MASTERY SKILLS CHECK: Chapter 12,	714
	CUMULATIVE SKILLS CHECK: Chapter 12,	715
	REVIEW SKILLS CHECK: Chapter 13,	717
13.1	EXERCISES,	717
13.2	EXERCISES,	719
13.3	EXERCISES,	720
<u>13.4</u>	EXERCISES,	723
13.5	EXERCISES,	724
13.7	EXERCISES,	724
	MASTERY SKILLS CHECK: Chapter 13,	726
	CUMULATIVE SKILLS CHECK: Chapter 13,	727
	REVIEW SKILLS CHECK: Chapter 14,	727
14.1	EXERCISES,	728
14.3	EXERCISES,	728
<u>14.4</u>	EXERCISES,	729
<u>14.5</u>	EXERCISES,	732
<u>14.6</u>	EXERCISES,	733
14.7	EXERCISES,	735
	MASTERY SKILLS CHECK: Chapter 14,	737

**Index . . . . . 739**

## Acknowledgments

I wish to say special thanks to

**Bjarne Stroustrup**  
**Steve Clamage**  
**P. J. Plauger**  
**Al Stevens**

for sharing their knowledge, advice, and expertise during the preparation of this book. It was much appreciated.



## Introduction

If you already know C and are moving up to C++, this book is for you.

C++ is the C programmer's answer to Object-Oriented Programming (OOP). Built upon the solid foundation of C, C++ adds support for OOP (and many other new features) without sacrificing any of C's power, elegance, or flexibility. C++ has become the universal language of programmers around the world and is the language that will create the next generation of high-performance software. It is the single most important language that a professional programmer must know.

C++ was invented in 1979 by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey. Initially it was called "C with classes." The name was changed to C++ in 1983. Since then, C++ has undergone three major revisions, the first in 1985 and the second in 1990. The third occurred during the C++ standardization process. Several years ago, work began on a standard for C++. Towards that end, a joint ANSI (American National Standards Institute) and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994. In that draft, the ANSI/ISO C++ committee (of which I am a member) kept the features first defined by Stroustrup and added some new ones as well. But, in general, this initial draft reflected the state of C++ at the time.

Soon after the completion of the first draft of the standard an event occurred that caused the standard to be greatly expanded: the creation of the Standard Template Library (STL) by Alexander Stepanov. As you will learn, the STL is a set of generic routines that you can use to manipulate data. It is both powerful and elegant. But it is also quite large. Subsequent to the first draft, the committee voted to include the STL in the specification for C++. The addition of the STL expanded the scope of C++ well beyond its original definition. While important, the inclusion of the STL, among other things, slowed the standardization of C++.

It is fair to say that the standardization of C++ took far longer than any one had expected when it began. However, it is now nearly complete. The final draft has been prepared and passed out of

committee. It now awaits only formal approval. In a practical sense, a standard for C++ is now a reality. Compilers already are beginning to support all of the new features.

The material in this book describes Standard C++. This is the version of C++ created by the ANSI/ISO standardization committee and it is the one that is currently accepted by all major compilers. Therefore, using this book, you can be confident that what you learn today will also apply tomorrow.

### ***What Is New in the Third Edition***

This is the third edition of *Teach Yourself C++*. It includes all of the material contained in the first two editions and adds two new chapters and many new topics. The first new chapter covers Run-Time Type ID (RTTI) and the new casting operators. The second covers the Standard Template Library (STL). Both of these topics are major features added to the C++ language since the previous edition was published. New topics include namespaces, the new-style headers, and coverage of the modern-style I/O system. In all, the third edition of *Teach Yourself C++* is substantially larger than its preceding two editions.

### ***If You're Using Windows***

If your computer uses Windows and your goal is to write Windows-based programs, then you have chosen the right language to learn. C++ is completely at home with Windows programming. However, none of the programs in this book are Windows programs. Instead, they are console-based programs. The reason for this is easy to understand: Windows programs are, by their nature, large and complex. The overhead required to create even a minimal Windows skeletal program is 50 to 70 lines of code. To write Windows programs that demonstrate the features of C++ would require hundreds of lines of code each. Put simply, Windows is not an appropriate environment in which to learn programming. However, you can still use a Windows-based compiler to compile the programs in this book because the compiler will automatically create a console session in which to execute your program.

Once you have mastered C++, you will be able to apply your knowledge to Windows programming. In fact, Windows programming

using C++ allows the use of class libraries such as MFC, that can greatly simplify the development of a Windows program.

### ***How This Book Is Organized***

This book is unique because it teaches you the C++ language by applying mastery learning. It presents one idea at a time, followed by numerous examples and exercises to help you master each topic. This approach ensures that you fully understand each topic before moving on.

The material is presented sequentially. Therefore, be sure to work carefully through the chapters. Each one assumes that you know the material presented in all preceding chapters. At the start of every chapter (except Chapter 1) there is a Review Skills Check that tests your knowledge of the preceding chapter. At the end of each chapter you will find a Mastery Skills Check that checks your knowledge of the material present in the chapter. Finally, each chapter concludes with a Cumulative Skills Check which tests how well you are integrating new material with that presented in earlier chapters. The answers to the book's many exercises are found in Appendix B.

This book assumes that you are already an accomplished C programmer. Put simply, you can't learn to program in C++ until you can program in C. If you can't program in C, take some time to learn it before attempting to use this book. A good way to learn C is to read my book *Teach Yourself C, Third Edition* (Osborne/McGraw-Hill, Berkeley CA, 1997). It uses the same presentation style as this book.

### ***Don't Forget: Code on the Web***

Remember, the source code for all of the programs in this book is available free-of-charge on the Web at <http://www.osborne.com>. Downloading this code prevents you from having to type in the examples.

## For Further Study

*Teach Yourself C++, Third Edition* is your gateway into the "Herb Schildt" series of programming books. Here is a partial list of Schildt's other books.

If you want to learn more about C++, then you will find these books especially helpful.

*C++: The Complete Reference*

*C++ From the Ground Up*

*Expert C++*

If you want to learn more about C, the foundation of C++, we recommend

*Teach Yourself C*

*C: The Complete Reference*

*The Annotated ANSI C Standard*

If you will be developing programs for the Web, you will want to read

*Java: The Complete Reference*

co-authored by Herbert Schildt and Patrick Naughton.

Finally, if you want to program for Windows, we recommend

*Schildt's Windows 95 Programming in C and C++*

*Schildt's Advanced Windows 95 Programming in C and C++*

*Windows NT 4 From the Ground Up*

*MFC Programming From the Ground Up*

**When you need solid answers, fast, turn to  
Herbert Schildt, the recognized authority  
on programming.**



# 1

## *An Overview of C++*

### **chapter objectives**

- 1.1** What is object-oriented programming?
- 1.2** Two versions of C++
- 1.3** C++ console I/O
- 1.4** C++ comments
- 1.5** Classes: A first look
- 1.6** Some differences between C and C++
- 1.7** Introducing function overloading
- 1.8** C++ keywords

---

**C**++ is an enhanced version of the C language. C++ includes everything that is part of C and adds support for object-oriented programming (OOP for short). In addition, C++ contains many improvements and features that simply make it a "better C," independent of object-oriented programming. With very few, very minor exceptions, C++ is a superset of C. While everything that you know about the C language is fully applicable to C++, understanding its enhanced features will still require a significant investment of time and effort on your part. However, the rewards of programming in C++ will more than justify the effort you put forth.

The purpose of this chapter is to introduce you to several of the most important features of C++. As you know, the elements of a computer language do not exist in a void, separate from one another. Instead, they work together to form the complete language. This interrelatedness is especially pronounced in C++. In fact, it is difficult to discuss one aspect of C++ in isolation because the features of C++ are highly integrated. To help overcome this problem, this chapter provides a brief overview of several C++ features. This overview will enable you to understand the examples discussed later in this book. Keep in mind that most topics will be more thoroughly explored in later chapters.

Since C++ was invented to support object-oriented programming, this chapter begins with a description of OOP. As you will see, many features of C++ are related to OOP in one way or another. In fact, the theory of OOP permeates C++. However, it is important to understand that C++ can be used to write programs that are and are *not* object oriented. How you use C++ is completely up to you.

At the time of this writing, the standardization of C++ is being finalized. For this reason, this chapter describes some important differences between versions of C++ that have been in common use during the past several years and the new Standard C++. Since this book teaches Standard C++, this material is especially important if you are using an older compiler.

In addition to introducing several important C++ features, this chapter also discusses some differences between C and C++ programming styles. There are several aspects of C++ that allow greater flexibility in the way that you write programs. While some of these features have little or nothing to do with object-oriented

programming, they are found in most C++ programs, so it is appropriate to discuss them early in this book.

Before you begin, a few general comments about the nature and form of C++ are in order. First, for the most part, C++ programs physically look like C programs. Like a C program, a C++ program begins execution at **main( )**. To include command-line arguments, C++ uses the same **argc**, **argv** convention that C uses. Although C++ defines its own, object-oriented library, it also supports all the functions in the C standard library. C++ uses the same control structures as C. C++ includes all of the built-in data types defined by C.

This book assumes that you already know the C programming language. In other words, you must be able to program in C before you can learn to program in C++ by using this book. If you don't know C, a good starting place is my book *Teach Yourself C, Third Edition* (Berkeley: Osborne/McGraw-Hill, 1997). It applies the same systematic approach used in this book and thoroughly covers the entire C language.



*This book assumes that you know how to compile and execute a program using your C++ compiler. If you don't, you will need to refer to your compiler's instructions. (Because of the differences between compilers, it is impossible to give compilation instructions for each in this book.) Since programming is best learned by doing, you are strongly urged to enter, compile, and run the examples in the book in the order in which they are presented.*

### 1.1

## **WHAT IS OBJECT-ORIENTED PROGRAMMING?**

Object-oriented programming is a powerful way to approach the task of programming. Since its early beginnings, programming has been governed by various methodologies. At each critical point in the evolution of programming, a new approach was created to help the programmer handle increasingly complex programs. The first programs were created by toggling switches on the front panel of the computer. Obviously, this approach is suitable for only the smallest programs. Next, assembly language was invented, which allowed longer programs to be written. The next advance happened in the 1950s when the first high-level language (FORTRAN) was invented.

By using a high-level language, a programmer was able to write programs that were several thousand lines long. However, the method

of programming used early on was an ad hoc, anything-goes approach. While this is fine for relatively short programs, it yields unreadable (and unmanageable) "spaghetti code" when applied to larger programs. The elimination of spaghetti code became feasible with the invention of *structured programming languages* in the 1960s. These languages include Algol and Pascal. In loose terms, C is a structured language, and most likely the type of programming you have been doing would be called structured programming. Structured programming relies on well-defined control structures, code blocks, the absence (or at least minimal use) of the GOTO, and stand-alone subroutines that support recursion and local variables. The essence of structured programming is the reduction of a program into its constituent elements. Using structured programming, the average programmer can create and maintain programs that are up to 50,000 lines long.

Although structured programming has yielded excellent results when applied to moderately complex programs, even it fails at some point, after a program reaches a certain size. To allow more complex programs to be written, a new approach to the job of programming was needed. Towards this end, object-oriented programming was invented. OOP takes the best of the ideas embodied in structured programming and combines them with powerful new concepts that allow you to organize your programs more effectively. Object-oriented programming encourages you to decompose a problem into its constituent parts. Each component becomes a self-contained object that contains its own instructions and data that relate to that object. In this way, complexity is reduced and the programmer can manage larger programs.

All OOP languages, including C++, share three common defining traits: encapsulation, polymorphism, and inheritance. Let's look at these concepts now.

## **ENCAPSULATION**

---

*Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

## POLYMORPHISM

*Polymorphism* (from the Greek, meaning "many forms") is the quality that allows one name to be used for two or more related but technically different purposes. As it relates to OOP, polymorphism allows one name to specify a general class of actions. Within a general class of actions, the specific action to be applied is determined by the type of data. For example, in C, which does not significantly support polymorphism, the absolute value action requires three distinct function names: **abs( )**, **labs( )**, and **fabs( )**. These functions compute and return the absolute value of an integer, a long integer, and a floating-point value, respectively. However, in C++, which supports polymorphism, each function can be called by the same name, such as **abs( )**. (One way this can be accomplished is shown later in this chapter.) The type of data used to call the function determines which specific version of the function is actually executed. As you will see, in C++, it is possible to use one function name for many different purposes. This is called *function overloading*.

More generally, the concept of polymorphism is characterized by the idea of "one interface, multiple methods," which means using a generic interface for a group of related activities. The advantage of polymorphism is that it helps to reduce complexity by allowing one interface to specify a *general class of action*. It is the compiler's job to select the *specific action* as it applies to each situation. You, the

programmer, don't need to do this selection manually. You need only remember and utilize the general interface. As the example in the preceding paragraph illustrates, having three names for the absolute value function instead of just one makes the general activity of obtaining the absolute value of a number more complex than it actually is.

Polymorphism can be applied to operators, too. Virtually all programming languages contain a limited application of polymorphism as it relates to the arithmetic operators. For example, in C, the + sign is used to add integers, long integers, characters, and floating-point values. In these cases, the compiler automatically knows which type of arithmetic to apply. In C++, you can extend this concept to other types of data that you define. This type of polymorphism is called *operator overloading*.

The key point to remember about polymorphism is that it allows you to handle greater complexity by allowing the creation of standard interfaces to related activities.

## **INHERITANCE**

*Inheritance* is the process by which one object can acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add those features that are specific only to itself. Inheritance is important because it allows an object to support the concept of *hierarchical classification*. Most information is made manageable by hierarchical classification. For example, think about the description of a house. A house is part of the general class called **building**. In turn, **building** is part of the more general class **structure**, which is part of the even more general class of objects that we call **man-made**. In each case, the child class inherits all those qualities associated with the parent and adds to them its own defining characteristics. Without the use of ordered classifications, each object would have to define all characteristics that relate to it explicitly. However, through inheritance, it is possible to describe an object by stating what general class (or classes) it belongs to along with those specific traits that make it unique. As you will see, inheritance plays a very important role in OOP.

**EXAMPLES**

1. Encapsulation is not entirely new to OOP. To a degree, encapsulation can be achieved when using the C language. For example, when you use a library function, you are, in effect, using a black-box routine, the internals of which you cannot alter or affect (except, perhaps, through malicious actions). Consider the **fopen( )** function. When it is used to open a file, several internal variables are created and initialized. As far as your program is concerned, these variables are hidden and not accessible. However, C++ provides a much more secure approach to encapsulation.
2. In the real world, examples of polymorphism are quite common. For example, consider the steering wheel on your car. It works the same whether your car uses power steering, rack-and-pinion steering, or standard, manual steering. The point is that the interface (the steering wheel) is the same no matter what type of actual steering mechanism (method) is used.
3. Inheritance of properties and the more general concept of classification are fundamental to the way knowledge is organized. For example, celery is a member of the **vegetable** class, which is part of the **plant** class. In turn, plants are living organisms, and so forth. Without hierarchical classification, systems of knowledge would not be possible.

**EXERCISE**

1. Think about the way that classification and polymorphism play an important role in our day-to-day lives.

At the time of this writing, C++ is in the midst of a transformation. As explained in the preface to this book, C++ has been undergoing the process of standardization for the past several years. The goal has been

to create a stable, standardized, feature-rich language that will suit the needs of programmers well into the next century. As a result, there are really two versions of C++. The first is the traditional version that is based upon Bjarne Stroustrup's original designs. This is the version of C++ that has been used by programmers for the past decade. The second is the new Standard C++, which was created by Stroustrup and the ANSI/ISO standardization committee. While these two versions of C++ are very similar at their core, Standard C++ contains several enhancements not found in traditional C++. Thus, Standard C++ is essentially a superset of traditional C++.

This book teaches Standard C++. This is the version of C++ defined by the ANSI/ISO standardization committee, and it is the version implemented by all modern C++ compilers. The code in this book reflects the contemporary coding style and practices as encouraged by Standard C++. This means that what you learn in this book will be applicable today as well as tomorrow. Put directly, Standard C++ is the future. And, since Standard C++ encompasses all features found in earlier versions of C++, what you learn in this book will enable you to work in all C++ programming environments.

However, if you are using an older compiler, it might not accept all of the programs in this book. Here's why: During the process of standardization, the ANSI/ISO committee added many new features to the language. As these features were defined, they were implemented by compiler developers. Of course, there is always a lag time between the addition of a new feature to the language and the availability of the feature in commercial compilers. Since features were added to C++ over a period of years, an older compiler might not support one or more of them. This is important because two recent additions to the C++ language affect every program that you will write—even the simplest. If you are using an older compiler that does not accept these new features, don't worry. There is an easy workaround, which is described in the following paragraphs.

The differences between old-style and modern code involve two new features: new-style headers and the **namespace** statement. To demonstrate these differences we will begin by looking at two versions of a minimal, do-nothing C++ program. The first version, shown here, reflects the way C++ programs were written until recently. (That is, it uses old-style coding.)

```
/*
   A traditional-style C++ program.
*/

#include <iostream.h>

int main()
{
    /* program code */
    return 0;
}
```

Since C++ is built on C, this skeleton should be largely familiar, but pay special attention to the **#include** statement. This statement includes the file **iostream.h**, which provides support for C++'s I/O system. (It is to C++ what **stdio.h** is to C.)

Here is the second version of the skeleton, which uses the modern style:

```
/*
   A modern-style C++ program that uses
   the new-style headers and a namespace.
*/
#include <iostream>
using namespace std;

int main()
{
    /* program code */
    return 0;
}
```

Notice the two lines in this program immediately after the first comment; this is where the changes occur. First, in the **#include** statement, there is no **.h** after the name **iostream**. And second, the next line, specifying a namespace, is new. Although both the new-style headers and namespaces will be examined in detail later in this book, a brief overview is in order now.

## THE NEW C++ HEADERS

As you know from your C programming experience, when you use a library function in a program, you must include its header file. This is

done using the **#include** statement. For example, in C, to include the header file for the I/O functions, you include **stdio.h** with a statement like this:

```
#include <stdio.h>
```

Here **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that the **#include** statement *includes a file*.

When C++ was first invented and for several years after that, it used the same style of headers as did C. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ has introduced a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that might be mapped to files by the compiler, but they need not be. The new-style C++ headers are abstractions that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style header is not a filename, it does not have a **.h** extension. Such a header consists solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++:

```
<iostream>
<fstream>
<vector>
<string>
```

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.

Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** and **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the standard C headers simply add a **c** prefix to the filename and drop the **.h**. For example, the new-style C++ header for **math.h** is **<cmath>**, and the one for **string.h** is **<cstring>**. Although it is currently permissible to include a C-style header file when using C library functions, this approach is

deprecated by Standard C++. (That is, it is not recommended.) For this reason, this book will use new-style C++ headers in all **#include** statements. If your compiler does not support new-style headers for the C function library, simply substitute the old-style, C-like headers.

Since the new-style header is a recent addition to C++, you will still find many, many older programs that don't use it. These programs instead use C-style headers, in which a filename is specified. As the old-style skeletal program shows, the traditional way to include the I/O header is as shown here:

```
#include <iostream.h>
```

This causes the file **iostream.h** to be included in your program. In general, an old-style header will use the same name as its corresponding new-style header with a **.h** appended.

As of this writing, all C++ compilers support the old-style headers. However, the old style headers have been declared obsolete, and their use in new programs is not recommended. This is why they are not used in this book.



*While still common in existing C++ code, old-style headers are obsolete.*

## NAMESPACES

When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. A *namespace* is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Traditionally, the names of library functions and other such items were simply placed into the global namespace (as they are in C). However, the contents of new-style headers are placed in the **std** namespace. We will look closely at namespaces later in this book. For now, you don't need to worry about them because you can use the statement

```
using namespace std;
```

to bring the **std** namespace into visibility (i.e., to put **std** into the global namespace). After this statement has been compiled, there is

no difference between working with an old-style header and a new-style one.

### ***WORKING WITH AN OLD COMPILER***

As mentioned, both namespaces and the new-style headers are recent additions to the C++ language. While virtually all new C++ compilers support these features, older compilers might not. If you have one of these older compilers, it will report one or more errors when it tries to compile the first two lines of the sample programs in this book. If this is the case, there is an easy workaround: simply use an old-style header and delete the **namespace** statement. That is, just replace

```
#include <iostream>
using namespace std;
```

with

```
#include <iostream.h>
```

This change transforms a modern program into a traditional-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace** statement.

One other point: For now and for the next few years, you will see many C++ programs that use the old-style headers and that do not include a **namespace** statement. Your C++ compiler will be able to compile them just fine. For new programs, however, you should use the modern style because it is the only style of program that complies with Standard C++. While old-style programs will continue to be supported for many years, they are technically noncompliant.

### ***EXERCISE***

1. Before proceeding, try compiling the new-style skeleton program shown above. Although it does nothing, compiling it will tell you if your compiler supports the modern C++ syntax. If it does not accept the new-style headers or the **namespace** statement, substitute the old-style header as described. Remember, if your compiler does not accept new-style code, you must make this change for each program in this book.

## 1.3

## C++ CONSOLE I/O

Since C++ is a superset of C, all elements of the C language are also contained in the C++ language. This implies that all C programs are also C++ programs by default. (Actually, there are some very minor exceptions to this rule, which are discussed later in this book.) Therefore, it is possible to write C++ programs that look just like C programs. While there is nothing wrong with this per se, it does mean that you will not be taking full advantage of C++. To get the maximum benefit from C++, you must write C++-style programs. This means using a coding style and features that are unique to C++.

Perhaps the most common C++-specific feature used by C++ programmers is its approach to console I/O. While you may still use functions such as **printf( )** and **scanf( )**, C++ provides a new, and better, way to perform these types of I/O operations. In C++, I/O is performed using *I/O operators* instead of I/O functions. The output operator is **<<** and the input operator is **>>**. As you know, in C, these are the left and right shift operators, respectively. In C++, they still retain their original meanings (left and right shift) but they also take on the expanded role of performing input and output. Consider this C++ statement:

```
cout << "This string is output to the screen.\n";
```

This statement causes the string to be displayed on the computer's screen. **cout** is a predefined stream that is automatically linked to the console when a C++ program begins execution. It is similar to C's **stdout**. As in C, C++ console I/O may be redirected, but for the rest of this discussion, it is assumed that the console is being used.

By using the **<<** output operator, it is possible to output any of C++'s basic types. For example, this statement outputs the value 100.99:

```
cout << 100.99;
```

In general, to output to the console, use this form of the **<<** operator:

```
cout << expression;
```

Here *expression* can be any valid C++ expression—including another output expression.

To input a value from the keyboard, use the `>>` input operator. For example, this fragment inputs an integer value into `num`:

```
int num;  
cin >> num;
```

Notice that `num` is *not* preceded by an `&`. As you know, when you use C's `scanf( )` function to input values, variables must have their addresses passed to the function so they can receive the values entered by the user. This is not the case when you are using C++'s input operator. (The reason for this will become clear as you learn more about C++.)

In general, to input values from the keyboard, use this form of `>>`:

```
cin >> variable;
```



*The expanded roles of `<<` and `>>` are examples of operator overloading.*

In order to use the C++ I/O operators, you must include the header `<iostream>` in your program. As explained earlier, this is one of C++'s standard headers and is supplied by your C++ compiler.

## EXAMPLES

1. This program outputs a string, two integer values, and a double floating-point value:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int i, j;  
    double d;  
  
    i = 10;  
    j = 20;  
    d = 99.101;
```

```
    cout << "Here are some values: ";
    cout << i;
    cout << ' ';
    cout << j;
    cout << ' ';
    cout << d;

    return 0;
}
```

The output of this program is shown here.

```
Here are some values: 10 20 99.101
```



*If you are working with an older compiler, it might not accept the new-style headers and the **namespace** statements used by this and other programs in this book. If this is the case, substitute the old-style code described in the preceding section.*

2. It is possible to output more than one value in a single I/O expression. For example, this version of the program described in Example 1 shows a more efficient way to code the I/O statements:

```
#include <iostream>
using namespace std;

int main()
{
    int i, j;
    double d;

    i = 10;
    j = 20;
    d = 99.101;

    cout << "Here are some values: ";
    cout << i << ' ' << j << ' ' << d;

    return 0;
}
```

Here the line

```
cout << i << ' ' << j << ' ' << d;
```

outputs several items in one expression. In general, you can use a single statement to output as many items as you like. If this seems confusing, simply remember that the `<<` output operator behaves like any other C++ operator and can be part of an arbitrarily long expression.

Notice that you must explicitly include spaces between items when needed. If the spaces are left out, the data will run together when displayed on the screen.

3. This program prompts the user for an integer value:

```
#include <iostream>
using namespace std;

int main()
{
    int i;

    cout << "Enter a value: ";
    cin >> i;
    cout << "Here's your number: " << i << "\n";

    return 0;
}
```

Here is a sample run:

```
Enter a value: 100
Here's your number: 100
```

As you can see, the value entered by the user is put into `i`.

4. The next program prompts the user for an integer value, a floating-point value, and a string. It then uses one input statement to read all three.

```
#include <iostream>
using namespace std;

int main()
{
    int i;
```

```
float f;
char s[80];

cout << "Enter an integer, float, and string: ";
cin >> i >> f >> s;
cout << "Here's your data: ";
cout << i << ' ' << f << ' ' << s;

return 0;
}
```

As this example illustrates, you can input as many items as you like in one input statement. As in C, individual data items must be separated by whitespace characters (spaces, tabs, or newlines).

When a string is read, input will stop when the first whitespace character is encountered. For example, if you enter the following into the preceding program

```
10 100.12 This is a test
```

the program will display this:

```
10 100.12 This
```

The string is incomplete because the reading of the string stopped with the space after **This**. The remainder of the string is left in the input buffer, awaiting a subsequent input operation. (This is similar to inputting a string by using **scanf( )** with the **%s** format.)

5. By default, when you use **>>**, all input is line buffered. This means that no information is passed to your C++ program until you press ENTER. (In C, the **scanf( )** function is line buffered, so this style of input should not be new to you.) To see the effect of line-buffered input, try this program:

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    cout << "Enter keys, x'to stop.\n";
```

```
    do {
        cout << " ";
        cin >> ch;
    } while (ch != 'x');

    return 0;
}
```

When you test this program, you will have to press ENTER after each key you type in order for the corresponding character to be sent to the program.

---

### EXERCISES

1. Write a program that inputs the number of hours that an employee works and the employee's wage. Then display the employee's gross pay. (Be sure to prompt for input.)
2. Write a program that converts feet to inches. Prompt the user for feet and display the equivalent number of inches. Have your program repeat this process until the user enters 0 for the number of feet.
3. Here is a C program. Rewrite it so it uses C++-style I/O statements.

```
/* Convert this C program into C++ style.
   This program computes the lowest common
   denominator.
*/
#include <stdio.h>

int main(void)
{
    int a, b, d, min;

    printf("Enter two numbers: ");
    scanf("%d%d", &a, &b);
    min = a > b ? b : a;
    for(d = 2; d<min; d++)
        if(((a%d)==0) && ((b%d)==0)) break;
```

```
if(d==min) {  
    printf("No common denominators\n");  
    return 0;  
}  
printf("The lowest common denominator is %d\n", d);  
return 0;  
}
```

## 1.4

## C++ COMMENTS

In C++, you can include comments in your program two different ways. First, you can use the standard, C-like comment mechanism. That is, begin a comment with `/*` and end it with `*/`. As with C, this type of comment cannot be nested in C++.

The second way that you can add a remark to your C++ program is to use the *single-line comment*. A single-line comment begins with `//` and stops at the end of the line. Other than the physical end of the line (that is, a carriage-return/linefeed combination), a single-line comment uses no comment terminator symbol.

Typically, C++ programmers use C-like comments for multiline commentaries and reserve C++-style single-line comments for short remarks.

### EXAMPLES

1. Here is a program that contains both C and C++-style comments:

```
/*  
 * This is a C-like comment.  
  
 * This program determines whether  
 * an integer is odd or even.  
 */  
  
#include <iostream>  
using namespace std;
```

```
int main()
{
    int num; // this is a C++ single-line comment

    // read the number
    cout << "Enter number to be tested: ";
    cin >> num;

    // see if even or odd
    if((num%2)==0) cout << "Number is even\n";
    else cout << "Number is odd\n";

    return 0;
}
```

2. While multiline comments cannot be nested, it is possible to nest a single-line comment within a multiline comment. For example, this is perfectly valid:

```
/* This is a multiline comment
   inside of which // is nested a single-line comment.
   Here is the end of the multiline comment.
*/
```

The fact that single-line comments can be nested within multiline comments makes it easier for you to "comment out" several lines of code for debugging purposes.

---

### EXERCISES

---

1. As an experiment, determine whether this comment (which nests a C-like comment within a C++-style, single-line comment) is valid:  
  
`// This is a strange /* way to do a comment */`
  2. On your own, add comments to the answers to the exercises in Section 1.3.
-

## 1.5

## CLASSES: A FIRST LOOK

Perhaps the single most important feature of C++ is the *class*. The class is the mechanism that is used to create objects. As such, the class is at the heart of many C++ features. Although the subject of classes is covered in great detail throughout this book, classes are so fundamental to C++ programming that a brief overview is necessary here.

A class is declared using the **class** keyword. The syntax of a **class** declaration is similar to that of a structure. Its general form is shown here:

```
class class-name {  
    // private functions and variables  
public:  
    // public functions and variables  
} object-list;
```

In a class declaration, the *object-list* is optional. As with a structure, you can declare class objects later, as needed. While the *class-name* is also technically optional, from a practical point of view it is virtually always needed. The reason for this is that the *class-name* becomes a new type name that is used to declare objects of the class.

Functions and variables declared inside a class declaration are said to be *members* of that class. By default, all functions and variables declared inside a class are private to that class. This means that they are accessible only by other members of that class. To declare public class members, the **public** keyword is used, followed by a colon. All functions and variables declared after the **public** specifier are accessible both by other members of the class and by any other part of the program that contains the class.

Here is a simple class declaration:

```
class myclass {  
    // private to myclass  
    int a;  
public:  
    void set_a(int num);  
    int get_a();  
};
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

type specifier. For example, this line declares two objects of type **myclass**:

```
myclass ob1, ob2; // these are objects of type myclass
```



*A class declaration is a logical abstraction that defines a new type. It determines what an object of that type will look like. An object declaration creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not.*

Once an object of a class has been created, your program can reference its public members by using the dot (period) operator in much the same way that structure members are accessed. Assuming the preceding object declaration, the following statement calls **set\_a()** for objects **ob1** and **ob2**:

```
ob1.set_a(10); // sets ob1's version of a to 10  
ob2.set_a(99); // sets ob2's version of a to 99
```

As the comments indicate, these statements set **ob1**'s copy of **a** to 10 and **ob2**'s copy to 99. Each object contains its own copy of all data declared within the class. This means that **ob1**'s **a** is distinct and different from the **a** linked to **ob2**.



*Each object of a class has its own copy of every variable declared within the class.*

## EXAMPLES

1. As a simple first example, this program demonstrates **myclass**, described in the text. It sets the value of **a** for **ob1** and **ob2** and then displays **a**'s value for each object:

```
#include <iostream>  
using namespace std;  
  
class myclass {  
    // private to myclass  
    int a;
```

```
public:  
    void set_a(int num);  
    int get_a();  
};  
  
void myclass::set_a(int num)  
{  
    a = num;  
}  
  
int myclass::get_a()  
{  
    return a;  
}  
  
int main()  
{  
    myclass ob1, ob2;  
  
    ob1.set_a(10);  
    ob2.set_a(99);  
  
    cout << ob1.get_a() << "\n";  
    cout << ob2.get_a() << "\n";  
    return 0;  
}
```

As you should expect, this program displays the values 10 and 99 on the screen.

2. In **myclass** from the preceding example, **a** is private. This means that only member functions of **myclass** can access it directly. (This is one reason why the public function **get\_a()** is required.) If you try to access a private member of a class from some part of your program that is not a member of that class, a compile-time error will result. For example, assuming that **myclass** is defined as shown in the preceding example, the following **main()** function will cause an error:

```
// This fragment contains an error.  
#include <iostream>  
using namespace std;  
  
int main()  
{
```

```
myclass ob1, ob2;

ob1.a = 10; // ERROR! cannot access private member
ob2.a = 99; // by non-member functions.

cout << ob1.get_a() << "\n";
cout << ob2.get_a() << "\n";

return 0;
}
```

3. Just as there can be public member functions, there can be public member variables as well. For example, if **a** were declared in the public section of **myclass**, **a** could be referenced by any part of the program, as shown here:

```
#include <iostream>
using namespace std;

class myclass {
public:
    // now a is public
    int a;
    // and there is no need for set_a() or get_a()
};

int main()
{
    myclass ob1, ob2;

    // here a is accessed directly
    ob1.a = 10;
    ob2.a = 99;

    cout << ob1.a << "\n";
    cout << ob2.a << "\n";

    return 0;
}
```

In this example, since **a** is declared as a public member of **myclass**, it is directly accessible from **main( )**. Notice how the dot operator is used to access **a**. In general, when you are calling a member function or accessing a member variable from outside

its class, the object's name followed by the dot operator followed by the member's name is required to fully specify which object's member you are referring to.

4. To get a taste of the power of objects, let's look at a more practical example. This program creates a class called **stack** that implements a stack that can be used to store characters:

```
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack

public:
    void init(); // initialize stack
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

// Initialize the stack
void stack::init()
{
    tos = 0;
}

// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Pop a character.
char stack::pop()
{
    if(tos-- 0) {
```

```
    cout << "Stack is empty";
    return 0; // return null on empty stack
}
tos--;
return stck[tos];
}

int main()
{
    stack s1, s2; // create two stacks
    int i;
    // initialize the stacks
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    return 0;
}
```

This program displays the following output:

```
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
```

Let's take a close look at this program now. The class **stack** contains two private variables: **stck** and **tos**. The array **stck** actually holds the characters pushed onto the stack, and **tos** contains the index to the top of the stack. The public stack functions are **init( )**, **push( )**, and **pop( )**, which initialize the stack, push a value, and pop a value, respectively.

Inside **main( )**, two stacks, **s1** and **s2**, are created, and three characters are pushed onto each stack. It is important to understand that each stack object is separate from the other. That is, the characters pushed onto **s1** in no way affect the characters pushed onto **s2**. Each object contains its own copy of **stck** and **tos**. This concept is fundamental to understanding objects. Although all objects of a class share their member functions, each object creates and maintains *its own data*.

---

**EXERCISES**

1. If you have not done so, enter and run the programs shown in the examples for this section.
  2. Create a class called **card** that maintains a library card catalog entry. Have the class store a book's title, author, and number of copies on hand. Store the title and author as strings and the number on hand as an integer. Use a public member function called **store( )** to store a book's information and a public member function called **show( )** to display the information. Include a short **main( )** function to demonstrate the class.
  3. Create a queue class that maintains a circular queue of integers. Make the queue size 100 integers long. Include a short **main( )** function that demonstrates its operation.
- 

1.6

**SOME DIFFERENCES BETWEEN C AND C++**

Although C++ is a superset of C, there are some small differences between the two, and a few are worth knowing from the start. Before proceeding, let's take time to examine them.

First, in C, when a function takes no parameters, its prototype has the word **void** inside its function parameter list. For example, in C, if a

function called **f1( )** takes no parameters (and returns a **char**), its prototype will look like this:

```
char f1(void);
```

However, in C++, the **void** is optional. Therefore, in C++, the prototype for **f1( )** is usually written like this:

```
char f1();
```

C++ differs from C in the way that an empty parameter list is specified. If the preceding prototype had occurred in a C program, it would simply mean that *nothing is said about* the parameters to the function. In C++, it means that the function *has no parameters*. This is the reason that the preceding examples did not explicitly use **void** to declare an empty parameters list. (The use of **void** to declare an empty parameter list is not illegal; it is just redundant. Since most C++ programmers pursue efficiency with a nearly religious zeal, you will almost never see **void** used in this way.) Remember, in C++, these two declarations are equivalent:

```
char f1();  
char f1(void);
```

Another subtle difference between C and C++ is that in a C++ program, all functions must be prototyped. Remember, in C, prototypes are recommended but technically optional. In C++, they are required. As the examples from the previous section show, a member function's prototype contained in a class also serves as its general prototype, and no other separate prototype is required.

A third difference between C and C++ is that in C++, if a function is declared as returning a value, it must return a value. That is, if a function has a return type other than **void**, any **return** statement within that function must contain a value. In C, a non-**void** function is not required to actually return a value. If it doesn't, a garbage value is "returned."

In C, if you don't explicitly specify the return type of a function, an integer return type is assumed. C++ has dropped the "default-to-int" rule. Thus, you must explicitly declare the return type of all functions.

One other difference between C and C++ that you will commonly encounter in C++ programs has to do with where local variables can be declared. In C, local variables can be declared only at the start of a

block, prior to any "action" statements. In C++, local variables can be declared anywhere. One advantage of this approach is that local variables can be declared close to where they are first used, thus helping to prevent unwanted side effects.

Finally, C++ defines the **bool** data type, which is used to store Boolean (i.e., true/false) values. C++ also defines the keywords **true** and **false**, which are the only values that a value of type **bool** can have. In C++, the outcome of the relational and logical operators is a value of type **bool**, and all conditional statements must evaluate to a **bool** value. Although this might at first seem to be a big change from C, it isn't. In fact, it is virtually transparent. Here's why: As you know, in C, **true** is any nonzero value and **false** is 0. This still holds in C++ because any nonzero value is automatically converted into **true** and any 0 value is automatically converted into **false** when used in a Boolean expression. The reverse also occurs: **true** is converted to 1 and **false** is converted to 0 when a **bool** value is used in an integer expression. The addition of **bool** allows more thorough type checking and gives you a way to differentiate between Boolean and integer types. Of course, its use is optional; **bool** is mostly a convenience.

## EXAMPLES

1. In a C program, it is common practice to declare **main( )** as shown here if it takes no command-line arguments:

```
int main(void)
```

However, in C++, the use of **void** is redundant and unnecessary.

2. This short C++ program will not compile because the function **sum( )** is not prototyped:

```
// This program will not compile.  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int a, b, c;  
  
    cout << "Enter two numbers: "
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



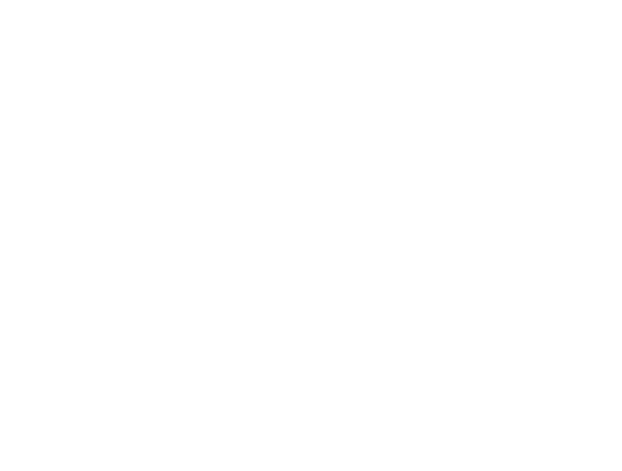
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    if(balance<0.0) cout << "****";
    cout << "\n";
}

int main()
{
    cl_type acc1(100.12, "Johnson");
    cl_type acc2(-12.34, "Hedricks");

    acc1.show();
    acc2.show();

    return 0;
}
```

2. Here is an example that uses a union to display the binary bit pattern, byte by byte, contained within a **double** value.

```
#include <iostream>
using namespace std;

union bits {
    bits(double n);
    void show_bits();
    double d;
    unsigned char c[sizeof(double)];
};

bits::bits(double n)
{
    d = n;
}

void bits::show_bits()
{
    int i, j;

    for(j = sizeof(double)-1; j>=0; j--) {
        cout << "Bit pattern in byte " << j << ": ";
        for(i = 128; i; i >>= 1)
            if(i & c[j]) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}
```

```
}

int main()
{
    bits ob(1991.829);

    ob.show_bits();

    return 0;
}
```

The output of this program is

```
Bit pattern in byte 7: 01000000
Bit pattern in byte 6: 10011111
Bit pattern in byte 5: 00011111
Bit pattern in byte 4: 01010000
Bit pattern in byte 3: 11100101
Bit pattern in byte 2: 01100000
Bit pattern in byte 1: 01000001
Bit pattern in byte 0: 10001001
```

3. Both structures and unions can have constructors and destructors. The following example shows the **strtype** class reworked as a structure. It contains both a constructor and a destructor function.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

struct strtype {
    strtype(char *ptr);
    ~strtype();
    void show();
private:
    char *p;
    int len;
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
```

```
if(!p) {
    cout << "Allocation error\n";
    exit(1);
}
strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Freeing p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("This is a test."), s2("I like C++.");
    s1.show();
    s2.show();

    return 0;
}
```

4. This program uses an anonymous union to display the individual bytes that comprise a **double**. (This program assumes that **doubles** are 8 bytes long.)

```
// Using an anonymous union.
#include <iostream>
using namespace std;

int main()
{
    union {
        unsigned char bytes[8];
        double value;
    };
    int i;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this example, **input( )** creates a local object called **str** and then reads a string from the keyboard. This string is copied into **str.s**, and then **str** is returned by the function. This object is then assigned to **ob** inside **main( )** when it is returned by the call to **input( )**.

2. You must be careful about returning objects from functions if those objects contain destructor functions because the returned object goes out of scope as soon as the value is returned to the calling routine. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is assigned the return value is still using it. For example, consider this incorrect version of the preceding program:

```
// An error generated by returning an object.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class samp {
    char *s;
public:
    samp() { s = '\0'; }
    ~samp() { if(s) free(s); cout << "Freeing s\n"; }
    void show() { cout << s << "\n"; }
    void set(char *str);
};

// Load a string.
void samp::set(char *str)
{
    s = (char *) malloc(strlen(str)+1);
    if(!s) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(s, str);
}

// Return an object of type samp.
samp input()
```

```
    {
        char s[80];
        samp str;

        cout << "Enter a string: ";
        cin >> s;

        str.set(s);
        return str;
    }

int main()
{
    samp ob;

    // assign returned object to ob
    ob = input(); // This causes an error!!!!
    ob.show();

    return 0;
}
```

The output from this program is shown here:

```
Enter a string: Hello
Freeing s
Freeing s
Hello
Freeing s
Null pointer assignment
```

Notice that **samp**'s destructor function is called three times. First, it is called when the local object **str** goes out of scope when **input( )** returns. The second time **~samp( )** is called is when the temporary object returned by **input( )** is destroyed. Remember, when an object is returned from a function, an invisible (to you) temporary object is automatically generated which holds the return value. In this case, this object is simply a copy of **str**, which is the return value of the function. Therefore, after the function has returned, the temporary object's destructor is executed. Finally, the destructor for object **ob**, inside **main( )**, is called when the program terminates.

The trouble is that in this situation, the first time the destructor executes, the memory allocated to hold the string

input by **input( )** is freed. Thus, not only do the other two calls to **samp**'s destructor try to free an already released piece of dynamic memory, but they destroy the dynamic allocation system in the process, as evidenced by the run-time message "Null pointer assignment." (Depending upon your compiler, the memory model used for compilation, and the like, you may or may not see this message if you try this program.)

The key point to be understood from this example is that when an object is returned from a function, the temporary object used to effect the return will have its destructor function called. Thus, you should avoid returning objects in which this situation is harmful. (As you will learn in Chapter 5, it is possible to use a copy constructor to manage this situation.)

---

## EXERCISES

---

1. To illustrate exactly when an object is constructed and destructed when returned from a function, create a class called **who**. Have **who**'s constructor take one character argument that will be used to identify an object. Have the constructor display a message similar to this when constructing an object:

Constructing who #x

where **x** is the identifying character associated with each object. When an object is destroyed, have a message similar to this displayed:

Destroying who #x

where, again, **x** is the identifying character. Finally, create a function called **make\_who( )** that returns a **who** object. Give each object a unique name. Note the output displayed by the program.

2. Other than the incorrect freeing of dynamically allocated memory, think of a situation in which it would be improper to return an object from a function.
-



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
// ...
strcpy(p, "This is a test");
```

Hint: A string is simply an array of characters.

2. Using **new**, show how to allocate a **double** and give it an initial value of -123.0987.
- 

## 4.6

## REFERENCES

C++ contains a feature that is related to the pointer: the *reference*. A reference is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used. First, a reference can be passed to a function. Second, a reference can be returned by a function. Finally, an independent reference can be created. Each of these applications of the reference is examined, beginning with reference parameters.

Without a doubt, the most important use of a reference is as a parameter to a function. To help you understand what a reference parameter is and how it works, let's first start with a program that uses a pointer (not a reference) as a parameter:

```
#include <iostream>
using namespace std;

void f(int *n); // use a pointer parameter

int main()
{
    int i = 0;

    f(&i);

    cout << "Here is i's new value: " << i << '\n';
}
```

```
    return 0;
}

void f(int *n)
{
    *n = 100; // put 100 into the argument pointed to by n
}
```

Here **f( )** loads the value 100 into the integer pointed to by **n**. In this program, **f( )** is called with the address of **i** in **main( )**. Thus, after **f( )** returns, **i** contains the value 100.

This program demonstrates how a pointer is used as a parameter to manually create a call-by-reference parameter-passing mechanism. In a C program, this is the only way to achieve a call-by-reference. However, in C++, you can completely automate this process by using a reference parameter. To see how, let's rework the previous program. Here is a version that uses a reference parameter:

```
#include <iostream>
using namespace std;

void f(int &n); // declare a reference parameter

int main()
{
    int i = 0;

    f(i); //

    cout << "Here is i's new value: " << i << '\n';

    return 0;
}

// f() now uses a reference parameter
void f(int &n)
{
    // notice that no * is needed in the following statement
    n = 100; // put 100 into the argument used to call f()
}
```

Examine this program carefully. First, to declare a reference variable or parameter, you precede the variable's name with the **&**. This is how **n** is declared as a parameter to **f( )**. Now that **n** is a

reference, it is no longer necessary—or even legal—to apply the `*` operator. Instead, each time `n` is used within `f()`, it is automatically treated as a pointer to the argument used to call `f()`. This means that the statement

```
n = 100;
```

actually puts the value 100 into the variable used to call `f()`, which, in this case, is `i`. Further, when `f()` is called, there is no need to precede the argument with the `&`. Instead, because `f()` is declared as taking a reference parameter, the address to the argument is *automatically* passed to `f()`.

To review, when you use a reference parameter, the compiler automatically passes the address of the variable used as the argument. There is no need to manually generate the address of the argument by preceding it with an `&` (in fact, it is not allowed). Further, within the function, the compiler automatically uses the variable pointed to by the reference parameter. There is no need to employ the `*` (and again, it is not allowed). Thus, a reference parameter fully automates the call-by-reference parameter-passing mechanism.

It is important to understand that you cannot change what a reference is pointing to. For example, if the statement

```
n++;
```

were put inside `f()` in the preceding program, `n` would still be pointing to `i` in `main()`. Instead of incrementing `n`, this statement increments the value of the variable being referenced (in this case, `i`).

Reference parameters offer several advantages over their (more or less) equivalent pointer alternatives. First, from a practical point of view, you no longer need to remember to pass the address of an argument. When a reference parameter is used, the address is automatically passed. Second, in the opinion of many programmers, reference parameters offer a cleaner, more elegant interface than the rather clumsy explicit pointer mechanism. Third, as you will see in the next section, when an object is passed to a function as a reference, no copy is made. This is one way to eliminate the troubles associated with the copy of an argument damaging something needed elsewhere

in  
the  
prog  
ram



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## 5.5

## OVERLOADING AND AMBIGUITY

When you are overloading functions, it is possible to introduce ambiguity into your program. Overloading-caused ambiguity can be introduced through type conversions, reference parameters, and default arguments. Further, some types of ambiguity are caused by the overloaded functions themselves. Other types occur in the manner in which an overloaded function is called. Ambiguity must be removed before your program will compile without error.

### EXAMPLES

1. One of the most common types of ambiguity is caused by C++'s automatic type conversion rules. As you know, when a function is called with an argument that is of a compatible (but not the same) type as the parameter to which it is being passed, the type of the argument is automatically converted to the target type. In fact, it is this sort of type conversion that allows a function such as **putchar( )** to be called with a character even though its argument is specified as an **int**. However, in some cases, this automatic type conversion will cause an ambiguous situation when a function is overloaded. To see how, examine this program:

```
// This program contains an ambiguity error.
#include <iostream>
using namespace std;

float f(float i)
{
    return i / 2.0;
}

double f(double i)
{
    return i / 3.0;
}

int main()
{
    float x = 10.09;
    double y = 10.09;
```

```
    cout << f(x); // unambiguous - use f(float)
    cout << f(y); // unambiguous - use f(double)

    cout << f(10); // ambiguous, convert 10 to double or float??

    return 0;
}
```

As the comments in `main()` indicate, the compiler is able to select the correct version of `f()` when it is called with either a **float** or a **double** variable. However, what happens when it is called with an integer? Does the compiler call **f(float)** or **f(double)**? (Both are valid conversions!) In either case, it is valid to promote an integer into either a **float** or a **double**. Thus, the ambiguous situation is created.

This example also points out that ambiguity can be introduced by the way an overloaded function is called. The fact is that there is no inherent ambiguity in the overloaded versions of `f()` as long as each is called with an unambiguous argument.

2. Here is another example of function overloading that is not ambiguous in and of itself. However, when this function is called with the wrong type of argument, C++'s automatic conversion rules cause an ambiguous situation.

```
// This program is ambiguous.
#include <iostream>
using namespace std;

void f(unsigned char c)
{
    cout << c;
}

void f(char c)
{
    cout << c;
}

int main()
{
    f('c');
    f(86); // which f() is called???
```

```
    return 0;  
}
```

Here, when **f( )** is called with the numeric constant 86, the compiler cannot know whether to call **f(unsigned char)** or **f(char)**. Either conversion is equally valid, thus leading to ambiguity.

3. One type of ambiguity is caused when you try to overload functions in which the only difference is the fact that one uses a reference parameter and the other uses the default call-by-value parameter. Given C++'s formal syntax, there is no way for the compiler to know which function to call. Remember, there is no syntactical difference between calling a function that takes a value parameter and calling a function that takes a reference parameter. For example:

```
// An ambiguous program.  
#include <iostream>  
using namespace std;  
  
int f(int a, int b)  
{  
    return a+b;  
}  
  
// this is inherently ambiguous  
int f(int a, int &b)  
{  
    return a-b;  
}  
  
int main()  
{  
    int x=1, y=2;  
  
    cout << f(x, y); // which version of f() is called???  
  
    return 0;  
}
```

Here, **f(x, y)** is ambiguous because it could be calling either version of the function. In fact, the compiler will flag an error before this statement is even specified because the overloading



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
// Output count number of spaces.  
void space(int count)  
{  
    for( ; count; count--) cout << ' ';  
}  
  
// Output count number of chs.  
void space(int count, char ch)  
{  
    for( ; count; count--) cout << ch;  
}  
  
int main()  
{  
    /* Create a pointer to void function with  
       one int parameter. */  
    void (*fp1)(int);  
  
    /* Create a pointer to void function with  
       one int parameter and one character parameter. */  
    void (*fp2)(int, char);  
  
    fp1 = space; // gets address of space(int)  
  
    fp2 = space; // gets address of space(int, char)  
  
    fp1(22); // output 22 spaces  
    cout << endl;  
  
    fp2(30, 'x'); // output 30 x's  
    cout << endl;  
  
    return 0;  
}
```

As the comments illustrate, the compiler is able to determine which overloaded function to obtain the address of based upon how **fp1** and **fp2** are declared.

To review: When you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is assigned. Further, the declaration of the function pointer must exactly match one and only one of the overloaded functions. If it does not, ambiguity will be introduced, causing a compile-time error.

**EXERCISE**

1. Following are two overloaded functions. Show how to obtain the address of each.

```
int dif(int a, int b)
{
    return a-b;
}

float dif(float a, float b)
{
    return a-b;
}
```

**SKILLS CHECK**

At this point you should be able to perform the following exercises and answer the questions.

1. Overload the **date( )** constructor from Section 5.1, Example 3, so that it accepts a parameter of type **time\_t**. (Remember, **time\_t** is a type defined by the standard time and date functions found in your C++ compiler's library.)
2. What is wrong with the following fragment?

```
class samp {
    int a;
public:
    samp(int i) { a = i; }
    // ...
};
```

```
// ...  
  
int main()  
{  
    samp x, y(10);  
  
    // ...  
}
```

3. Give two reasons why you might want (or need) to overload a class's constructor.
4. What is the most common general form of a copy constructor?
5. What type of operations will cause the copy constructor to be invoked?
6. Briefly explain what the **overload** keyword does and why it is no longer needed.
7. Briefly describe a default argument.
8. Create a function called **reverse( )** that takes two parameters. The first parameter, called **str**, is a pointer to a string that will be reversed upon return from the function. The second parameter is called **count**, and it specifies how many characters of **str** to reverse. Give **count** a default value that, when present, tells **reverse( )** to reverse the entire string.
9. What is wrong with the following prototype?

```
char *wordwrap(char *str, int size=0, char ch);
```

10. Explain some ways that ambiguity can be introduced when you are overloading functions.
11. What is wrong with the following fragment?

```
void compute(double *num, int divisor=1);  
void compute(double *num);  
// ...  
compute(&x);
```

12. When you are assigning the address of an overloaded function to a pointer, what is it that determines which version of the function is used?



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Create a function called **order( )** that takes two integer reference parameters. If the first argument is greater than the second argument, reverse the two arguments. Otherwise, take no action. That is, order the two arguments used to call **order( )** so that, upon return, the first argument will be less than the second. For example, given

```
int x=1, y=0;  
order(x, y);
```

following the call, **x** will be 0 and **y** will be 1.

2. Why are the following two overloaded functions inherently ambiguous?

```
int f(int a);  
int f(int &a);
```

3. Explain why using a default argument is related to function overloading.
4. Given the following partial class, add the necessary constructor functions so that both declarations within **main( )** are valid.  
(Hint: You need to overload **samp( )** twice.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

passed to the derived class's constructor in the normal fashion. However, if you need to pass an argument to the constructor of the base class, a little more effort is needed. To accomplish this, a chain of argument passing is established. First, all necessary arguments to both the base class and the derived class are passed to the derived class's constructor. Using an expanded form of the derived class's constructor declaration, you then pass the appropriate arguments along to the base class. The syntax for passing along an argument from the derived class to the base class is shown here:

```
derived-constructor (arg-list) : base(arg-list) {  
    // body of derived class constructor  
}
```

Here *base* is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is also possible for the derived class to ignore all arguments and just pass them along to the base.

## EXAMPLES

1. Here is a very short program that illustrates when base class and derived class constructor and destructor functions are executed:

```
#include <iostream>  
using namespace std;  
  
class base {  
public:  
    base() { cout << "Constructing base class\n"; }  
    ~base() { cout << "Destructing base class\n"; }  
};  
  
class derived : public base {  
public:  
    derived() { cout << "Constructing derived class\n"; }  
    ~derived() { cout << "Destructing derived class\n"; }  
};  
  
int main()  
{
```

```
    derived o;

    return 0;
}
```

This program displays the following output:

```
Constructing base class
Constructing derived class
Destructing derived class
Destructing base class
```

As you can see, the constructors are executed in order of derivation and the destructors are executed in reverse order.

2. This program shows how to pass an argument to a derived class's constructor:

```
#include <iostream>
using namespace std;

class base {
public:
    base() { cout << "Constructing base class\n"; }
    ~base() { cout << "Destructing base class\n"; }
};

class derived : public base {
    int j;
public:
    derived(int n) {
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showj();

    return 0;
}
```

Notice that the argument is passed to the derived class's constructor in the normal fashion.

3. In the following example, both the derived class and the base class constructors take arguments. In this specific case, both use the same argument, and the derived class simply passes along the argument to the base.

```
#include <iostream>
using namespace std;

class base {
    int i;
public:
    base(int n) {
        cout << "Constructing base class\n";
        i = n;
    }
    ~base() { cout << "Destructing base class\n"; }
    void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n) : base(n) { // pass arg to base class
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showi();
    o.showj();

    return 0;
}
```

Pay special attention to the declaration of **derived**'s constructor. Notice how the parameter **n** (which receives the initialization argument) is both used by **derived( )** and passed to **base( )**.

4. In most cases, the constructor functions for the base and derived classes will *not* use the same argument. When this is the case and you need to pass one or more arguments to each, you must pass to the derived class's constructor *all* arguments needed by *both* the derived class and the base class. Then the derived class simply passes along to the base those arguments required by it. For example, this program shows how to pass an argument to the derived class's constructor and another one to the base class:

```
#include <iostream>
using namespace std;

class base {
    int i;
public:
    base(int n) {
        cout << "Constructing base class\n";
        i = n;
    }
    ~base() { cout << "Destructing base class\n"; }
    void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n, int m) : base(m) { // pass arg to base class
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10, 20);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
// Pop a character.  
char stack::pop()  
{  
    if(tos==0) {  
        cout << "Stack is empty\n";  
        return 0; // return null on empty stack  
    }  
    tos--;  
    return stck[tos];  
}
```

2. Write a program that contains a class called **watch**. Using the standard time functions, have this class's constructor read the system time and store it. Create an inserter that displays the time.
3. Using the following class, which converts feet to inches, create an extractor that prompts the user for feet. Also, create an inserter that displays the number of feet and inches. Include a program that demonstrates that your inserter and extractor work.

```
class ft_to_inches {  
    double feet;  
    double inches;  
public:  
    void set(double f) {  
        feet = f;  
        inches = f * 12;  
    }  
};
```



Copyrighted material



# 9

## *Advanced C++ I/O*

### **chapter objectives**

- 9.1** Creating your own manipulators
- 9.2** File I/O basics
- 9.3** Unformatted, binary I/O
- 9.4** More unformatted I/O functions
- 9.5** Random access
- 9.6** Checking the I/O status
- 9.7** Customized I/O and files



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

7. What predefined streams are created when a C++ program begins execution?

9.1

## CREATING YOUR OWN MANIPULATORS

In addition to overloading the insertion and extraction operators, you can further customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons. First, a manipulator can consolidate a sequence of several separate I/O operations. For example, it is not uncommon to have situations in which the same sequence of I/O operations occurs frequently within a program. In these cases you can use a custom manipulator to perform these actions, thus simplifying your source code and preventing accidental errors. Second, a custom manipulator can be important when you need to perform I/O operations on a nonstandard device. For example, you could use a manipulator to send control codes to a special type of printer or an optical recognition system.

Custom manipulators are a feature of C++ that supports OOP, but they can also benefit programs that aren't object oriented. As you will see, custom manipulators can help make any I/O-intensive program clearer and more efficient.

As you know, there are two basic types of manipulators: those that operate on input streams and those that operate on output streams. In addition to these two broad categories, there is a secondary division: those manipulators that take an argument and those that don't. There are some significant differences between the way a parameterless manipulator and a parameterized manipulator are created. Further, creating parameterized manipulators is substantially more difficult than creating parameterless ones and is beyond the scope of this book. However, writing your own parameterless manipulators is quite easy and is examined here.

All parameterless manipulator output functions have this skeleton:

```
ostream &manip-name(ostream &stream)
{
    // your code here
    return stream;
}
```

Here *manip-name* is the name of the manipulator and *stream* is a reference to the invoking stream. A reference to the stream is returned. This is necessary if a manipulator is used as part of a larger I/O expression. It is important to understand that even though the manipulator has as its single argument a reference to the stream upon which it is operating, no argument is used when the manipulator is called in an output operation.

All parameterless input manipulator functions have this skeleton:

```
istream &manip-name(istream &stream)
{
    // your code here
    return stream;
}
```

An input manipulator receives a reference to the stream on which it was invoked. This stream must be returned by the manipulator.



*It is crucial that your manipulators return a reference to the invoking stream. If this is not done, your manipulators cannot be used in a sequence of input or output operations.*

## EXAMPLES

- As a simple first example, the following program creates a manipulator called **setup( )** that sets the field width to 10, the precision to 4, and the fill character to \*.

```
#include <iostream>
using namespace std;

ostream &setup(ostream &stream)
{
    stream.width(10);
    stream.precision(4);
    stream.fill('*');

    return stream;
}

int main()
```

```
{  
    cout << setup << 123.123456;  
  
    return 0;  
}
```

As you can see, **setup** is used as part of an I/O expression in the same way that any of the built-in manipulators would be used.

2. Custom manipulators need not be complex to be useful. For example, the simple manipulators **atn( )** and **note( )**, shown here, provide a shorter way to output frequently used words or phrases.

```
#include <iostream>  
using namespace std;  
  
// Attention:  
ostream &atn(ostream &stream)  
{  
    stream << "Attention: ";  
    return stream;  
}  
  
// Please note:  
ostream &note(ostream &stream)  
{  
    stream << "Please Note: ";  
    return stream;  
}  
  
int main()  
{  
    cout << atn << "High voltage circuit\n";  
    cout << note << "Turn off all lights\n";  
  
    return 0;  
}
```

Even though they are simple, if used frequently, these manipulators save you from some tedious typing.

3. This program creates the **getpass( )** input manipulator, which rings the bell and then prompts for a password:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



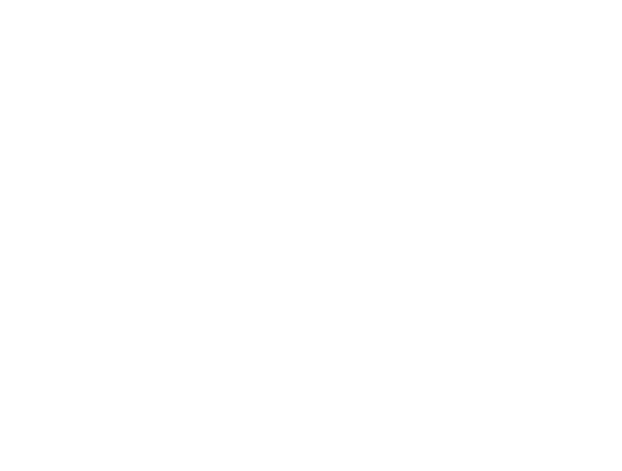
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



# 10

## *Virtual Functions*

### **chapter objectives**

- 10.1** Pointers to derived classes
- 10.2** Introduction to virtual functions
- 10.3** More about virtual functions
- 10.4** Applying polymorphism

**T**HIS chapter examines another important aspect of C++: the virtual function. What makes virtual functions important is that they are used to support run-time polymorphism. Polymorphism is supported by C++ in two ways. First, it is supported at compile time, through the use of overloaded operators and functions. Second, it is supported at run time, through the use of virtual functions. As you will learn, run-time polymorphism provides the greatest flexibility.

At the foundation of virtual functions and run-time polymorphism are pointers to derived classes. For this reason this chapter begins with a discussion of such pointers.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Create a manipulator that causes numbers to be displayed in scientific notation, using a capital E.
2. Write a program that copies a text file. During the copy process, convert all tabs into the correct number of spaces.
3. Write a program that searches a text file for a word specified on the command line. Have the program display how many times the specified word is found. For simplicity, assume that anything surrounded by whitespace is a word.
4. Show the statement that sets the put pointer to the 234th byte in a file linked to a stream called **out**.
5. What functions report status information about the C++ I/O system?
6. Give one advantage of using the C++ I/O functions instead of the C-like I/O system.

## 10.1

## POINTERS TO DERIVED CLASSES

Although Chapter 4 discussed C++ pointers at some length, one special aspect was deferred until now because it relates specifically to virtual functions. The feature is this: A pointer declared as a pointer to a base class can also be used to point to any class derived from that base. For example, assume two classes called **base** and **derived**, where **derived** inherits **base**. Given this situation, the following statements are correct:

```
base *p; // base class pointer

base base_ob; // object of type base
derived derived_ob; // object of type derived

// p can, of course, point to base objects
p = &base_ob; // p points to base object

// p can also point to derived objects without error
p = &derived_ob; // p points to derived object
```

As the comments suggest, a base pointer can point to an object of any class derived from that base without generating a type mismatch error.

Although you can use a base pointer to point to a derived object, you can access only those members of the derived object that were inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the members added by the derived class.

While it is permissible for a base pointer to point to a derived object, the reverse is not true. A pointer of the derived type cannot be used to access an object of the base class. (A type cast can be used to overcome this restriction, but its use is not recommended practice.)

One final point: Remember that pointer arithmetic is relative to the data type the pointer is declared as pointing to. Thus, if you point a base pointer to a derived object and then increment that pointer, it will not be pointing to the next derived object. It will be pointing to (what it thinks is) the next base object. Be careful about this.

**EXAMPLE**

1. Here is a short program that illustrates how a base class pointer can be used to access a derived class:

```
// Demonstrate pointer to derived class.  
#include <iostream>  
using namespace std;  
  
class base {  
    int x;  
public:  
    void setx(int i) { x = i; }  
    int getx() { return x; }  
};  
  
class derived : public base {  
    int y;  
public:  
    void sety(int i) { y = i; }  
    int gety() { return y; }  
};  
  
int main()  
{  
    base *p; // pointer to base type  
    base b_ob; // object of base  
    derived d_ob; // object of derived  
  
    // use p to access base object  
    p = &b_ob;  
    p->setx(10); // access base object  
    cout << "Base object x: " << p->getx() << '\n';  
  
    // use p to access derived object  
    p = &d_ob; // point to derived object  
    p->setx(99); // access derived object  
  
    // can't use p to set y, so do it directly  
    d_ob.sety(88);  
    cout << "Derived object x: " << p->getx() << '\n';  
    cout << "Derived object y: " << d_ob.gety() << '\n';  
  
    return 0;  
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
p = &t;  
cout << "Triangle has area: " << p->getarea() << '\n';  
  
return 0;  
}
```

Notice that the definition of **getarea( )** inside **area** is just a placeholder and performs no real function. Because **area** is not linked to any specific type of figure, there is no meaningful definition that can be given to **getarea( )** inside **area**. In fact, **getarea( )** must be overridden by a derived class in order to be useful. In the next section, you will see a way to enforce this.

---

## EXERCISES

1. Write a program that creates a base class called **num**. Have this class hold an integer value and contain a virtual function called **shownum( )**. Create two derived classes called **outhex** and **outoct** that inherit **num**. Have the derived classes override **shownum( )** so that it displays the value in hexadecimal and octal, respectively.
  2. Write a program that creates a base class called **dist** that stores the distance between two points in a **double** variable. In **dist**, create a virtual function called **trav\_time( )** that outputs the time it takes to travel that distance, assuming that the distance is in miles and the speed is 60 miles per hour. In a derived class called **metric**, override **trav\_time( )** so that it outputs the travel time assuming that the distance is in kilometers and the speed is 100 kilometers per hour.
- 

As Example 4 from the preceding section illustrates, sometimes when a virtual function is declared in the base class there is no meaningful operation for it to perform. This situation is common because often a

base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived class supplies the remainder. When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class *must* override this function. To ensure that this will occur, C++ supports *pure virtual functions*.

A pure virtual function has no definition relative to the base class. Only the function's prototype is included. To make a pure virtual function, use this general form:

```
virtual type func-name( parameter-list ) = 0;
```

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guarantee that a derived class will provide its own redefinition.

When a class contains at least one pure virtual function, it is referred to as an *abstract class*. Since an abstract class contains at least one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited. They are neither intended nor able to stand alone. It is important to understand, however, that you can still create a pointer to an abstract class, since it is through the use of base class pointers that run-time polymorphism is achieved. (It is also permissible to have a reference to an abstract class.)

When a virtual function is inherited, so is its virtual nature. This means that when a derived class inherits a virtual function from a base class and then the derived class is used as a base for yet another derived class, the virtual function can be overridden by the final derived class (as well as the first derived class). For example, if base class B contains a virtual function called `f()`, and D1 inherits B and D2 inherits D1, both D1 and D2 can override `f()` relative to their respective classes.

**EXAMPLES**

1. Here is an improved version of the program shown in Example 4 in the preceding section. In this version, the function **getarea()** is declared as pure in the base class **area**.

```
// Create an abstract class.  
#include <iostream>  
using namespace std;  
  
class area {  
    double dim1, dim2; // dimensions of figure  
public:  
    void setarea(double d1, double d2)  
    {  
        dim1 = d1;  
        dim2 = d2;  
    }  
    void getdim(double &d1, double &d2)  
    {  
        d1 = dim1;  
        d2 = dim2;  
    }  
    virtual double getarea() = 0; // pure virtual function  
};  
  
class rectangle : public area {  
public:  
    double getarea()  
    {  
        double d1, d2;  
  
        getdim(d1, d2);  
        return d1 * d2;  
    }  
};  
  
class triangle : public area {
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    }
}

int main()
{
    cout << "start\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "end";

    return 0;
}
```

The output produced by this program is shown here:

```
start
Caught 0
Caught One!
Caught One!
end
```

As this example suggests, using **catch(...)** as a default is a good way to catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

3. The following program shows how to restrict the types of exceptions that can be thrown from a function:

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
}

int main()
```

```
{  
    cout << "start\n";  
  
    try{  
        Xhandler(0); // also, try passing 1 and 2 to Xhandler()  
    }  
    catch(int i) {  
        cout << "Caught int\n";  
    }  
    catch(char c) {  
        cout << "Caught char\n";  
    }  
    catch(double d) {  
        cout << "Caught double\n";  
    }  
  
    cout << "end";  
  
    return 0;  
}
```

In this program, the function **Xhandler( )** can throw only integer, character, and **double** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, **unexpected( )** will be called.) To see an example of this, remove **int** from the list and retry the program.

It is important to understand that a function can only be restricted in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a function can throw any type of exception so long as it is caught *within* that function. The restriction applies only when throwing an exception out of the function.

4. The following change to **Xhandler( )** prevents it from throwing any exceptions:

```
// This function can throw NO exceptions!  
void Xhandler(int test) throw()  
{  
    /* The following statements no longer work. Instead,  
       they will cause an abnormal program termination. */  
    if(test==0) throw test;  
    if(test==1) throw 'a';
```

This program displays the following output:

```
start
Caught char * inside Xhandler
Caught char * inside main
end
```

---

### EXERCISES

1. Before continuing, compile and run all of the examples in this section. Be sure you understand why each program produces the output that it does.
2. What is wrong with this fragment?

```
try {
    // ...
    throw 10;
}
catch(int *p) {
    // ...
}
```

3. Show one way to fix the preceding fragment.
4. What **catch** expression catches all types of exceptions?
5. Here is a skeleton for a function called **divide( )**.

```
double divide(double a, double b)
{
    // add error handling
    return a/b;
}
```

This function returns the result of dividing **a** by **b**. Add error checking to this function using C++ exception handling. Specifically, prevent a divide-by-zero error. Demonstrate your solution in a program.

---



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



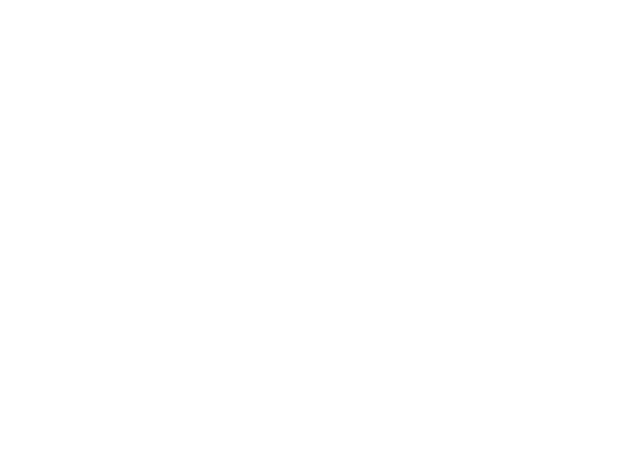
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    return NULL;
}

int main()
{
    int i;
    Shape *p;

    for(i=0; i<10; i++) {
        p = generator(); // get next object

        cout << typeid(*p).name() << endl;

        // draw object only if it is not a NullShape
        if(typeid(*p) != typeid(NullShape))
            p->example();
    }

    return 0;
}
```

Sample output from the program is shown here.

```
class Rectangle
*****
*
*
*****
class NullShape
class Triangle
*
*
*
*****
class Line
*****
class Rectangle
*****
*
*
*****
class Line
*****
```

```
class Triangle
*
* *
* *
*****
class Triangle
*
* *
* *
*****
class Triangle
*
* *
* *
*****
class Line
*****
```

5. The **typeid** operator can be applied to template classes. For example, consider the following program. It creates a hierarchy of template classes that store a value. The virtual function **get\_val( )** returns a value that is defined by each class. For class **Num**, this is the value of the number itself. For **Square**, it is the square of the number, and for **Sqr\_root**, it is the square root of the number. Objects derived from **Num** are generated by the **generator( )** function. The **typeid** operator is used to determine what type of object has been generated.

```
// typeid can be used with templates.
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i) { x = i; }
    virtual T get_val() { return x; }
};

template <class T>
class Square : public Num<T> {
```

```

p1 = generator(); // get next object

    if(typeid(*p1) == typeid(Square<double>))
        cout << "Square object: ";
    if(typeid(*p1) == typeid(Sqr_root<double>))
        cout << "Sqr_root object: ";

    cout << "Value is: " << p1->get_val();
    cout << endl;
}

return 0;
}

```

The output from the program is shown here.

```

class Num<double>
class Square<double>
class Sqr_root<double>
is Square<double>
Value is: 10000

```

Now, generate some Objects.

```

Sqr_root object: Value is: 8.18535
Square object: Value is: 0
Sqr_root object: Value is: 4.89898
Square object: Value is: 3364
Square object: Value is: 4096
Sqr_root object: Value is: 6.7082
Sqr_root object: Value is: 5.19615
Sqr_root object: Value is: 9.53939
Sqr_root object: Value is: 6.48074
Sqr_root object: Value is: 6

```

## EXERCISES

1. Why is RTTI a necessary feature of C++?
2. Try the experiment described in Example 1. What output do you see?
3. Is the following fragment correct?

```
cout << typeid(float).name();
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    if(p2) cout << "Square object: ";
    p3 = dynamic_cast<Sqr_root<double> *> (p1);
    if(p3) cout << "Sqr_root object: ";

    cout << "Value is: " << p1->get_val();
    cout << endl;
}

return 0;
}
```

---

## EXERCISES

1. In your own words, explain the purpose of **dynamic\_cast**.
2. Given the following fragment and using **dynamic\_cast**, show how you can assign **p** a pointer to some object **ob** if and only if **ob** is a **D2** object.

```
class B {
    virtual void f() {}
};

class D1: public B {
    void f() {}
};

class D2: public B {
    void f() {}
};

B *p;
```
3. Convert the **main( )** function in Section 12.1, Exercise 4, so that it uses **dynamic\_cast** rather than **typeid** to prevent a **NullShape** object from being displayed.
4. Using the **Num** class hierarchy from Example 3 in this section, will the following work?

```
Num<int> *Bp;
Square<double> *Dp;
// ...
Dp = dynamic_cast<Num<int>> (Bp);
```

---

```
void f(const double &i)
{
    i = 100; // Error -- fix using const_cast
}

int main()
{
    double x = 98.6;

    cout << x << endl;
    f(x);
    cout << x << endl;

    return 0;
}
```

3. Explain why **const\_cast** should normally be reserved for special cases.

---

## SKILLS CHECK



At this point you should be able to perform the following exercises and answer the questions.

1. Describe the operation of **typeid**.
2. What header must you include in order to use **typeid**?
3. In addition to the standard cast, C++ defines four casting operators. What are they and what are they for?
4. Complete the following partial program so that it reports which type of object has been selected by the user.

```
#include <iostream>
#include <typeinfo>
using namespace std;

class A {
    virtual void f() {}
};

class B : public A {
};

class C: public B {
};

int main()
{
    A *p, a_ob;
    B b_ob;
    C c_ob;
    int i;

    cout << "Enter 0 for A objects, ";
    cout << "1 for B objects or ";
    cout << "2 for C objects.\n";

    cin >> i;

    if(i==1) p = &b_ob;
    else if(i==2) p = &c_ob;
    else p = &a_ob;

    // report type of object selected by user

    return 0;
}
```

5. Explain how **dynamic\_cast** can sometimes be an alternative to **typeid**.
6. What type of object is obtained by the **typeid** operator?



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Rework the program in Section 12.1, Example 4 so that it uses exception handling to watch for an allocation failure within the **generator( )** function.
2. Change the **generator( )** function from Question 1 so that it uses the **nothrow** version of **new**. Be sure to check for errors.
3. Special Challenge: On your own, create a class hierarchy that has at its top an abstract class called **DataStruct**. Create two concrete subclasses. Have one implement a stack, the other a queue. Create a function called **DataStructFactory( )** that has this prototype:

```
DataStruct *DataStructFactory(char what);
```

Have **DataStructFactory( )** create a stack if *what* is s and a queue if *what* is q. Return a pointer to the object created. Demonstrate that your factory function works.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
class myclass {
    int a;
public:
    myclass(int x) { a = x; }
    myclass(char *str) { a = atoi(str); }
    int geta() { return a; }
};

int main()
{
    myclass ob1 = 4;      // converts to myclass(4)
    myclass ob2 = "123"; // converts to myclass("123");

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    return 0;
}
```

Since both constructors use different type arguments (as, of course, they must), each initialization statement is automatically converted into its equivalent constructor call.

2. The automatic conversion from the type of a constructor's first argument into a call to the constructor itself has interesting implications. For example, assuming **myclass** from Example 1, the following **main( )** function makes use of the conversions from **int** and **char \*** to assign **ob1** and **ob2** new values.

```
int main()
{
    myclass ob1 = 4;      // converts to myclass(4)
    myclass ob2 = "123"; // converts to myclass("123");

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    // use automatic conversions to assign new values
    ob1 = "1776"; // converts into ob1 = myclass("1776");
    ob2 = 2001;   // converts into ob2 = myclass(2001);

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;
}
```

```
    return 0;
}
```

3. To prevent the conversions just shown, you could specify the constructors as **explicit**, as shown here:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
    int a;
public:
    explicit myclass(int x) { a = x; }
    explicit myclass(char *str) { a = atoi(str); }
    int geta() { return a; }
};
```

---

### EXERCISES

1. In Example 3, if only **myclass(int)** is made explicit, will **myclass(char \*)** still allow implicit conversions? (Hint: Try it.)
2. Will the following fragment work?

```
class Demo {
    double x
public:
    Demo(double i) { x = i; }
    // ...
};

// ...
Demo counter = 10;
```

3. Justify the inclusion of the **explicit** keyword. (In other words, explain why implicit constructor conversions might not be a desirable feature of C++ in some cases.)
-

## 13.6

## USING LINKAGE SPECIFIERS AND THE *asm* KEYWORD

C++ provides two important mechanisms that make it easier to link C++ to other languages. One is the *linkage specifier*, which tells the compiler that one or more functions in your C++ program will be linked with another language that might have a different approach to naming, parameter passing, stack restoration, and the like. The second is the **asm** keyword, which allows you to embed assembly language instructions in your C++ source code. Both are examined here.

By default, all functions in a C++ program are compiled and linked as C++ functions. However, you can tell the C++ compiler to link a function so that it is compatible with another type of language. All C++ compilers allow functions to be linked as either C or C++ functions. Some also allow you to link functions with languages such as Pascal, Ada, or FORTRAN. To cause a function to be linked for a different language, use this general form of the linkage specification:

```
extern "language" function-prototype;
```

Here *language* is the name of the language with which you want the specified function to link. If you want to specify linkage for more than one function, use this form of the linkage specification:

```
extern "language" {  
    function-prototypes  
}
```

All linkage specifications must be global; they cannot be used inside a function.

The most common use of linkage specifications occurs when linking C++ programs to C code. By specifying "C" linkage you prevent the compiler from *mangling* (also known as *decorating*) the names of functions with embedded type information. Because of C++'s ability to overload functions and create member functions, the link-name of a function usually has type information added to it. Since C does not support overloading or member functions, it cannot recognize a mangled name. Using "C" linkage avoids this problem.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Member Function**

```
void splice(iterator i,
           list<T, Allocator> &ob,
           iterator e);
void splice(iterator i,
           list<T, Allocator> &ob,
           iterator start, iterator end);
void swap(list<T, Allocator> &ob)
void unique();
template <class BinPred>
void unique(BinPred pr);
```

**Description**

Removes the element pointed to by *e* from the list *ob* and stores it in the invoking list at the location pointed to by *i*.

Removes the range defined by *start* and *end* from *ob* and stores it in the invoking list beginning at the location pointed to by *i*.

Exchanges the elements stored in the invoking list with those in *ob*.

Removes duplicate elements from the invoking list. The second form uses *pr* to determine uniqueness.

TABLE 14-3 The *list* Member Functions (continued) ▼**EXAMPLES**

1. Here is a simple example of a list.

```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> lst; // create an empty list
    int i;

    for(i=0; i<10; i++) lst.push_back('A'+i);

    cout << "Size = " << lst.size() << endl;

    list<char>::iterator p;

    cout << "Contents: ";
    while(!lst.empty()) {
        p = lst.begin();
        cout << *p;
        lst.pop_front();
    }
}
```

4. You can sort a list by calling the **sort( )** member function. The following program creates a list of random characters and then puts the list into sorted order.

```
// Sort a list.  
#include <iostream>  
#include <list>  
#include <cstdlib>  
using namespace std;  
  
int main()  
{  
    list<char> lst;  
    int i;  
  
    // create a list of random characters  
    for(i=0; i<10; i++)  
        lst.push_back('A'+ (rand()%26));  
  
    cout << "Original contents: ";  
    list<char>::iterator p = lst.begin();  
    while(p != lst.end()) {  
        cout << *p;  
        p++;  
    }  
    cout << endl << endl;  
  
    // sort the list  
    lst.sort();  
    cout << "Sorted contents: ";  
    p = lst.begin();  
    while(p != lst.end()) {  
        cout << *p;  
        p++;  
    }  
  
    return 0;  
}
```

Here is sample output produced by the program.

```
Original contents: PHQGHUMEAY  
Sorted contents: AEGHHMPQUY
```

5. One ordered list can be merged with another. The result is an ordered list that contains the contents of the two original lists. The new list is left in the invoking list and the second list is left empty. This example merges two lists. The first contains the letters ACEGI and the second BDFHJ. These lists are then merged to produce the sequence ABCDEFGHIJ.

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> lst1, lst2;
    int i;

    for(i=0; i<10; i+=2) lst1.push_back('A'+i);
    for(i=1; i<11; i+=2) lst2.push_back('A'+i);

    cout << "Contents of lst1: ";
    list<char>::iterator p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;

    cout << "Contents of lst2: ";
    p = lst2.begin();
    while(p != lst2.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;

    // now, merge the two lists
    lst1.merge(lst2);
    if(lst2.empty())
        cout << "lst2 is now empty\n";

    cout << "Contents of lst1 after merge:\n";
    p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The sequences to be merged are defined by *start1*, *end1* and *start2*, *end2*. The result is put into the sequence pointed to by *result*. An iterator to the end of the resulting sequence is returned. Demonstrate this algorithm.

## 14.7

## THE STRING CLASS

As you know, C++ does not support a built-in string type, per se. It does, however, provide two ways to handle strings. First, you can use the traditional, null-terminated character array with which you are already familiar. This is sometimes referred to as a *C string*. The second method is to use a class object of type **string**. This is the approach that is examined here.

Actually, the **string** class is a specialization of a more general template class called **basic\_string**. In fact, there are two specializations of **basic\_string**: **string**, which supports 8-bit character strings, and **wstring**, which supports wide character strings. Since 8-bit characters are by far the most commonly used characters in normal programming, **string** is the version of **basic\_string** examined here.

Before you look at the **string** class, it is important that you understand why it is part of the C++ library. Standard classes have not been casually added to C++. In fact, a significant amount of thought and debate has accompanied each new addition. Given that C++ already contains some support for strings as null-terminated character arrays, it might at first seem that the inclusion of the **string** class is an exception to this rule. However, this is actually far from the truth. Here is why: Null-terminated strings cannot be manipulated by any of the standard C++ operators, nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];  
  
s1 = "one"; // can't do  
s2 = "two"; // can't do  
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings. These operations must be written using library functions, as shown here:

```
strcpy(s1, "one");
strcpy(s2, "two");
strcpy(s3, s1);
strcat(s3, s2);
```

Since null-terminated character arrays are not technically data types in their own right, the C++ operators cannot be applied to them. This makes even the most rudimentary string operations clumsy. More than anything else, it is the inability to operate on null-terminated strings using the standard C++ operators that has driven the development of a standard **string** class. Remember, when you define a class in C++, you are defining a new data type that can be fully integrated into the C++ environment. This, of course, means that the operators can be overloaded relative to the new class. Therefore, with the addition of a standard **string** class, it becomes possible to manage strings in the same way that any other type of data is managed: through the use of operators.

There is, however, one other reason for the standard **string** class: safety. An inexperienced or careless programmer can very easily overrun the end of an array that holds a null-terminated string. For example, consider the standard string copy function **strcpy( )**. This function contains no provision for checking the boundary of the target array. If the source array contains more characters than the target array can hold, a program error or system crash is possible (likely). As you will see, the standard **string** class prevents such errors.

In the final analysis, there are three reasons for the inclusion of the standard **string** class: consistency (a string now defines a data type), convenience (you can use the standard C++ operators), and safety (array boundaries will not be overrun). Keep in mind that there is no reason that you should abandon normal, null-terminated strings altogether. They are still the most efficient way in which to implement strings. However, when speed is not an overriding concern, the new **string** class gives you access to a safe and fully integrated way to manage strings.

Although not traditionally thought of as part of the STL, **string** is another container class defined by C++. This means that it supports the algorithms described in the previous section. However, strings have additional capabilities. To have access to the **string** class you must include `<string>` in your program.

The **string** class is very large, with many constructors and member functions. Also, many member functions have multiple overloaded forms. For this reason, it is not possible to look at the entire contents of **string** in this chapter. Instead, we will examine several of its most commonly used features. Once you have a general understanding of how **string** works, you will be able to easily explore the rest of it on your own.

The **string** class supports several constructors. The prototypes for three of its most commonly used constructors are shown here.

```
string();
string(const char *str);
string(const string &str);
```

The first form creates an empty **string** object. The second creates a **string** object from the null-terminated string pointed to by *str*. This form provides a conversion from null-terminated strings to **string** objects. The third form creates a **string** object from another **string** object.

A number of operators that apply to strings are defined for **string** objects, including those listed in the following table:

<u>Operator</u>	<u>Meaning</u>
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[]	Subscripting

<u>Operator</u>	<u>Meaning</u>
<<	Output
>>	Input

These operators allow the use of **string** objects in normal expressions and eliminate the need for calls to functions such as **strcpy( )** or **strcat( )**, for example. In general, you can mix **string** objects with normal, null-terminated strings in expressions. For example, a **string** object can be assigned a null-terminated string.

The **+** operator can be used to concatenate a **string** object with another **string** object or a **string** object with a C-style string. That is, the following variations are supported:

string + string  
string + C-string  
C-string + string

The **+** operator can also be used to concatenate a character onto the end of a string.

The **string** class defines the constant **npos**, which is usually **-1**. This constant represents the length of the longest possible string.

Although most simple string operations can be accomplished with the string operators, more complex or subtle ones are accomplished with **string** class member functions. Although there are far too many to discuss in this chapter, we will examine several of the most common ones. To assign one string to another, use the **assign( )** function. Two of its forms are shown here:

**string &assign(const string &strob, size\_type start, size\_type num);**

**string &assign(const char \*str, size\_type num);**

In the first form, *num* characters from *strob* beginning at the index specified by *start* will be assigned to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are assigned to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the **=** operator to assign one entire string to another. You will need to use the **assign( )** function only when assigning a partial string.

You can append part of one string to another using the **append( )** member function. Two of its forms are shown here:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
}

void card::show()
{
    cout << "Title: " << title << "\n";
    cout << "Author: " << author << "\n";
    cout << "Number on hand: " << number << "\n";
}

int main()
{
    card book1, book2, book3;

    book1.store("Dune", "Frank Herbert", 2);
    book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
    book3.store("The Rainbow", "D. H. Lawrence", 1);

    book1.show();
    book2.show();
    book3.show();

    return 0;
}

3. #include <iostream>
using namespace std;

#define SIZE 100

class q_type {
    int queue[SIZE]; // holds the queue
    int head, tail; // indices of head and tail
public:
    void init(); // initialize
    void q(int num); // store
    int dq(); // retrieve
};

// Initialize
void q_type::init()
{
    head = tail = 0;
}

// Put value on a queue.
```

```
void q_type::q(int num)
{
    if(tail+1==head || (tail+1==SIZE && !head)) {
        cout << "Queue is full\n";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // cycle around
    queue[tail] = num;
}

// Remove a value from a queue.
int q_type::deq()
{
    if(head == tail) {
        cout << "Queue is empty\n";
        return 0; // or some other error indicator
    }
    head++;
    if(head==SIZE) head = 0; // cycle around
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    q1.init();
    q2.init();

    for(i=1; i<=10; i++) {
        q1.q(i);
        q2.q(i*i);
    }

    for(i=1; i<=10; i++) {
        cout << "Dequeue 1: " << q1.deq() << "\n";
        cout << "Dequeue 2: " << q2.deq() << "\n";
    }

    return 0;
}
```

**1.6*****EXERCISES***

1. The function `f()` is not prototyped.

**1.7*****EXERCISES***

```
1. #include <iostream>
#include <cmath>
using namespace std;

// Overload sroot() for integers, longs, and doubles.

int sroot(int i);
long sroot(long i);
double sroot(double i);

int main()
{
    cout << "Square root of 90.34 is : " << sroot(90.34);
    cout << "\n";
    cout << "Square root of 90L is : " << sroot(90L);
    cout << "\n";
    cout << "Square root of 90 is : " << sroot(90);

    return 0;
}

// Return square root of integer.
int sroot(int i)
{
    cout << "computing integer root\n";
    return (int) sqrt((double) i);
}

// Return square root of long.
long sroot(long i)
{
    cout << "computing long root\n";
    return (long) sqrt((double) i);
}

// Return square root of double.
double sroot(double i)
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    cout << "Rectangle: " << b.area() << "\n";
    cout << "Triangle: " << i.area() << "\n";

    return 0;
}
```

**2.5*****EXERCISES***

1. // Stack class using a structure.

```
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters using a structure.
struct stack {
    stack(); // constructor
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
private:
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
};

// Initialize the stack.
stack::stack()
{
    cout << "Constructing a stack\n";
    tos = 0;
}

// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}
```

```
// Pop a character.  
char stack::pop()  
{  
    if(tos==0) {  
        cout << "Stack is empty\n";  
        return 0; // return null on empty stack  
    }  
    tos--;  
    return stck[tos];  
}  
  
int main()  
{  
    // Create two stacks that are automatically initialized.  
    stack s1, s2;  
    int i;  
  
    s1.push('a');  
    s2.push('x');  
    s1.push('b');  
    s2.push('y');  
    s1.push('c');  
    s2.push('z');  
  
    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";  
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";  
  
    return 0;  
}  
  
2. #include <iostream>  
using namespace std;  
  
union swapbytes {  
    unsigned char c[2];  
    unsigned i;  
    swapbytes(unsigned x);  
    void swp();  
};  
  
swapbytes::swapbytes(unsigned x)  
{  
    i = x;  
}
```

```

void swapbytes::swp()
{
    unsigned char temp;

    temp = c[0];
    c[0] = c[1];
    c[1] = temp;
}

int main()
{
    swapbytes ob(1);

    ob.swp();
    cout << ob.i;

    return 0;
}

```

3. An anonymous union is the syntactic mechanism that allows two variables to share the same memory space. The members of an anonymous union are accessed directly, without reference to an object. They are at the same scope level as the union itself.

**2.6*****EXERCISES***

```

1. #include <iostream>
using namespace std;

// Overload abs() three ways:

// abs() for ints
inline int abs(int n)
{
    cout << "In integer abs()\n";
    return n<0 ? -n : n;
}

// abs() for longs
inline long abs(long n)
{
    cout << "In long abs()\n";
    return n<0 ? -n : n;
}

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

4. You can expand a function in line either by preceding its definition with the **inline** specifier or by including its definition within a class declaration.
5. An in-line function must be defined before it is first used. Other common restrictions include the following: It cannot contain any loops. It must not be recursive. It cannot contain a **goto** or a **switch** statement. It cannot contain any **static** variables.
6. sample ob(100, 'x');

## 3.1

## EXERCISES

1. The assignment statement **x = y** is wrong because **cl1** and **cl2** are two different types of classes, and objects of differing class types cannot be assigned.

```
2. #include <iostream>
using namespace std;

#define SIZE 100

class q_type {
    int queue[SIZE]; // holds the queue
    int head, tail; // indices of head and tail
public:
    q_type(); // constructor
    void q(int num); // store
    int deq(); // retrieve
};

// Constructor
q_type::q_type()
{
    head = tail = 0;
}

// Put value on queue.
void q_type::q(int num)
{
    if(tail+1==head || (tail+1==SIZE && !head)) {
        cout << "Queue is full\n";
        return;
    }
}
```

```
tail++;
if(tail==SIZE) tail = 0; // cycle around
queue[tail] = num;
}

// Remove value from queue.
int q_type::deq()
{
    if(head == tail) {
        cout << "Queue is empty\n";
        return 0; // or some other error indicator
    }
    head++;
    if(head==SIZE) head = 0; // cycle around
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    for(i=1; i<=10; i++) {
        q1.q(i);
    }

    // assign one queue to another
    q2 = q1;

    // show that both have the same contents
    for(i=1; i<=10; i++)
        cout << "Dequeue 1: " << q1.deq() << "\n";

    for(i=1; i<=10; i++)
        cout << "Dequeue 2: " << q2.deq() << "\n";

    return 0;
}
```

3. If memory to hold a queue is dynamically allocated, assigning one queue to another causes the dynamic memory allocated to the queue on the left side of the assignment statement to be lost and the memory allocated to the queue on the right side to be freed twice when the objects are destroyed. Either of these two conditions is an unacceptable error.

**3.2****EXERCISES**

```
1. #include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters.
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
public:
    stack(); // constructor
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack()
{
    cout << "Constructing a stack\n";
    tos = 0;
}

// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Pop a character.
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}
```

```
class pr2; // forward declaration

class pr1 {
    int printing;
    // ...
public:
    pr1() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
    friend int inuse(pr1 o1, pr2 o2);
};

class pr2 {
    int printing;
    // ...
public:
    pr2() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
    friend int inuse(pr1 o1, pr2 o2);
};

// Return true if printer is in use.
int inuse(pr1 o1, pr2 o2)
{
    if(o1.printing || o2.printing) return 1;
    else return 0;
}

int main()
{
    pr1 p1;
    pr2 p2;

    if(!inuse(p1, p2)) cout << "Printer idle\n";

    cout << "Setting p1 to printing...\n";
    p1.set_print(1);
    if(inuse(p1, p2)) cout << "Now, printer in use.\n";

    cout << "Turn off p1...\n";
    p1.set_print(0);
    if(!inuse(p1, p2)) cout << "Printer idle\n";

    cout << "Turn on p2...\n";
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
void summation::set_sum(int n)
{
    int i;

    num = n;

    sum = 0;
    for(i=1; i<=n; i++)
        sum += i;
}

summation make_sum()
{
    int i;
    summation temp;

    cout << "Enter number: ";
    cin >> i;

    temp.set_sum(i);

    return temp;
}

int main()
{
    summation s;

    s = make_sum();

    s.show_sum();

    return 0;
}
```

6. For some compilers, in-line functions cannot contain loops.

7. #include <iostream>  
using namespace std;  
  
class myclass {
 int num;
public:
 myclass(int x) { num = x; }
 friend int isneg(myclass ob);

```
};

int isneg(myclass ob)
{
    return (ob.num < 0) ? 1 : 0;
}

int main()
{
    myclass a(-1), b(2);

    cout << isneg(a) << ' ' << isneg(b);
    cout << "\n";

    return 0;
}
```

8. Yes, a friend function can be friends with more than one class.

**4.1****EXERCISES**

```
1. #include <iostream>
using namespace std;

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};

int main()
{
    letters ob[10] = { 'a', 'b', 'c', 'd', 'e', 'f',
                      'g', 'h', 'i', 'j' };

    int i;

    for(i=0; i<10; i++)
        cout << ob[i].get_ch() << ' ';

    cout << "\n";
```

```
{  
    letters ob[10] = {  
        letters('a'),  
        letters('b'),  
        letters('c'),  
        letters('d'),  
        letters('e'),  
        letters('f'),  
        letters('g'),  
        letters('h')  
        letters('i'),  
        letters('j')  
    };  
  
    int i;  
  
    for(i=0; i<10; i++)  
        cout << ob[i].get_ch() << ' ';  
  
    cout << "\n";  
  
    return 0;  
}
```

**4.2****EXERCISES**

1. // Display in reverse order.

```
#include <iostream>  
using namespace std;  
  
class samp {  
    int a, b;  
public:  
    samp(int n, int m) { a = n; b = m; }  
    int get_a() { return a; }  
    int get_b() { return b; }  
};  
  
int main()  
{  
    samp ob[4] = {  
        samp(1, 2),  
        samp(3, 4),  
        samp(5, 6),
```

```
    for(i=0; i<4; i++) {
        cout << p->get_a() << ' ';
        p++;
        cout << p->get_a() << "\n";
        p++;
    }

    cout << "\n";

    return 0;
}
```

**4.3****EXERCISE**

```
1. // Use this pointer.

#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    myclass(int n, int m) { this->a = n; this->b = m; }
    int add() { return this->a + this->b; }
    void show();
};

void myclass::show()
{
    int t;

    t = this->add(); // call member function
    cout << t << "\n";
}

int main()
{
    myclass ob(10, 14);

    ob.show();

    return 0;
}
```

```
void phone::store(char *n, char *num)
{
    strcpy(name, n);
    strcpy(number, num);
}

void phone::show()
{
    cout << name << ":" << number;
    cout << "\n";
}

int main()
{
    phone *p;

    p = new phone;

    if(!p) {
        cout << "Allocation error.";
        return 1;
    }

    p->store("Isaac Newton", "111 555-2323");

    p->show();

    delete p;

    return 0;
}
```

3. On failure, **new** will either return a null pointer or generate an exception. You must check your compiler's documentation to determine which approach is used. In Standard C++, **new** generates an exception by default.

**4.5*****EXERCISES***

1. `char *p;`  
`p = new char [100];`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

int l;

l = strlen(s)+1;

p = new char [l];
if(!p) {
    cout << "Allocation error\n";
    exit(1);
}

strcpy(p, s);
}

// Fix by using a reference parameter.
void show(strtype &x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}

```

**4.8****EXERCISES**

1. // A simple bounded two-dimensional array example.  

```

#include <iostream>
#include <cstdlib>
using namespace std;

class array {
    int isize, jsize;
    int *p;
public:
    array(int i, int j);

```

```
    for(j=0; j<3; j++)
        cout << a.get(i, j) << ' ';
    // generate out of bounds
    a.put(10, 10);

    return 0;
}
```

2. No. A reference returned by a function cannot be assigned to a pointer.

## **MASTERY SKILLS CHECK: CHAPTER 4**

```
1. #include <iostream>
using namespace std;

class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << ' ' << b << '\n'; }
};

int main()
{
    a_type ob[2][5] = {
        a_type(1, 1), a_type(2, 2),
        a_type(3, 3), a_type(4, 4),
        a_type(5, 5), a_type(6, 6),
        a_type(7, 7), a_type(8, 8),
        a_type(9, 9), a_type(10, 10)
    };

    int i, j;

    for(i=0; i<2; i++)
        for(j=0; j<5; j++)
            ob[i][j].show();
}
```

```
        cout << '\n';

        return 0;
    }

2. #include <iostream>
using namespace std;

class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << ' ' << b << '\n'; }
};

int main()
{
    a_type ob[2][5] = {
        a_type(1, 1), a_type(2, 2),
        a_type(3, 3), a_type(4, 4),
        a_type(5, 5), a_type(6, 6),
        a_type(7, 7), a_type(8, 8),
        a_type(9, 9), a_type(10, 10)
    };

    a_type *p;

    p = (a_type *) ob;

    int i, j;

    for(i=0; i<2; i++)
        for(j=0; j<5; j++) {
            p->show();
            p++;
        }

    cout << '\n';

    return 0;
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    watch.start();
    for(i=0; i<320000; i++) ; // time a for loop
    watch.stop();
    watch.show();

    // create object using initial value
    stopwatch s2(clock());
    for(i=0; i<250000; i++) ; // time a for loop
    s2.stop();
    s2.show();

    return 0;
}
```

## 5.2

## EXERCISES

1. The **obj** and **temp** objects are constructed normally. However, when **temp** is returned by **f( )**, a temporary object is made, and it is this temporary object that generates the call to the copy constructor.
2. As the program is written, when an object is passed to **getval( )** a bitwise copy is made. When **getval( )** returns and that copy is destroyed, the memory allocated to that object (which is pointed to by **p**) is released. However, this is the same memory still required by the object used in the call to **getval( )**. The correct version of the program is shown here. It uses a copy constructor to avoid this problem.

```
// This program is now fixed.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
    int *p;
public:
    myclass(int i);
    myclass(const myclass &o); // copy constructor
    ~myclass() { delete p; }
    friend int getval(myclass o);
};
```

```
myclass::myclass(int i)
{
    p = new int;

    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    *p = i;
}

// Copy constructor
myclass::myclass(const myclass &o)
{
    p = new int; // allocate copy's own memory

    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    *p = *o.p;
}

int getval(myclass o)
{
    return *o.p; // get value
}

int main()
{
    myclass a(1), b(2);

    cout << getval(a) << " " << getval(b);
    cout << "\n";
    cout << getval(a) << " " << getval(b);

    return 0;
}
```

3. A copy constructor is invoked when one object is used to initialize another. A normal constructor is called when an object is created.

```
void myclreol(int len = -1);

int main()
{
    int i;

    gotoxy(1, 1);
    for(i=0; i<24; i++)
        cout << "abcdefghijklmnopqrstuvwxyz1234567890\n";

    gotoxy(1, 2);
    myclreol();
    gotoxy(1, 4);
    myclreol(20);

    return 0;
}
// Clear to end of line unless len parameter is specified.
void myclreol(int len)
{
    int x, y;

    x = wherex(); // get x position
    y = wherey(); // get y position

    if(len == -1) len = 80-x;

    int i = x;

    for( ; i<=len; i++) cout << ' ';

    gotoxy(x, y); // reset the cursor
}
```

4. A default argument cannot be another parameter or a local variable.

**5.6****EXERCISE**

```
1. #include <iostream>
using namespace std;

int dif(int a, int b)
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
}

void reverse(char *str, int count)
{
    int i, j;
    char temp;

    if(!count) count = strlen(str)-1;

    for(i=0, j=count; i<j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
```

9. All parameters receiving default arguments must appear to the right of those that do not.
10. Ambiguity can be introduced by default type conversions, reference parameters, and default arguments.
11. It is ambiguous because the compiler cannot know which version of **compute( )** to call. Is it the first version, with **divisor** defaulting? Or is it the second version, which takes only one parameter?
12. When you are obtaining the address of an overloaded function, it is the type specification of the pointer that determines which function is used.

## CUMULATIVE SKILLS CHECK: Chapter 5

```
1. #include <iostream>
using namespace std;

void order(int &a, int &b)
{
    int t;

    if(a<b) return;
    else { // swap a and b
        t = a;
        a = b;
        b = t;
    }
}
```

**6.2****EXERCISES**

```
1. // Overload the * and / relative to coord class.  
#include <iostream>  
using namespace std;  
  
class coord {  
    int x, y; // coordinate values  
public:  
    coord() { x=0; y=0; }  
    coord(int i, int j) { x=i; y=j; }  
    void get_xy(int &i, int &j) { i=x; j=y; }  
    coord operator*(coord ob2);  
    coord operator/(coord ob2);  
};  
  
// Overload * relative to coord class.  
coord coord::operator*(coord ob2)  
{  
    coord temp;  
  
    temp.x = x * ob2.x;  
    temp.y = y * ob2.y;  
  
    return temp  
}  
  
// Overload / relative to coord class.  
coord coord::operator/(coord ob2)  
{  
    coord temp;  
  
    temp.x = x / ob2.x;  
    temp.y = y / ob2.y;  
  
    return temp  
}  
  
int main()  
{  
    coord o1(10, 10), o2(5, 3), o3;  
    int x, y;  
  
    o3 = o1 * o2;
```

```
coord o1(10, 10), o2(5, 3);

if(o1>o2) cout << "o1 > o2\n";
else cout << "o1 <= o2 \n";

if(o1<o2) cout << "o1 < o2\n";
else cout << "o1 >= o2\n";

return 0;
}
```

## 6.4

**EXERCISES**

1. // Overload the -- relative to coord class.

```
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator--(); // prefix
    coord operator--(int notused); // postfix
};

// Overload prefix -- for coord class.
coord coord::operator--()
{
    x--;
    y--;
    return *this;
}

// Overload postfix -- for coord class.
coord coord::operator--(int notused)
{
    x--;
    y--;
    return *this;
}

int main()
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        return *this;
    }

int main()
{
    coord o1(10, 10), o2(-2, -2);
    int x, y;
    o1 = o1 + o2; // addition
    o1.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o2 = +o2; // absolute value
    o2.get_xy(x, y);
    cout << "(+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

**6.5****EXERCISES**

1. /\* Overload the - and / relative to coord class using friend functions. \*/  

```
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator-(coord ob1, coord ob2);
    friend coord operator/(coord ob1, coord ob2);
};

// Overload - relative to coord class using friend.
coord operator-(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x - ob2.x;
    temp.y = ob1.y - ob2.y;

    return temp;
}
```

```
class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator--(coord &ob); // prefix
    friend coord operator--(coord &ob, int notused); // postfix
};

// Overload -- (prefix) for coord class using a friend.
coord operator--(coord &ob)
{
    ob.x--;
    ob.y--;
    return ob;
}

// Overload -- (postfix) for coord class using a friend.
coord operator--(coord &ob, int notused)
{
    ob.x--;
    ob.y--;
    return ob;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    --o1; // decrement an object
    o1.get_xy(x, y);
    cout << "(--o1) X: " << x << ", Y: " << y << "\n";

    o1--; // decrement an object
    o1.get_xy(x, y);
    cout << "(o1--) X: " << x << " Y: " << y << "\n";

    return 0;
}
```

**6.6****EXERCISE**

```
1. #include <iostream>
   #include <cstdlib>
   using namespace std;

   class dynarray {
       int *p;
       int size;
   public:
       dynarray(int s);
       int &put(int i);
       int get(int i);
       dynarray &operator=(dynarray &ob);
   };

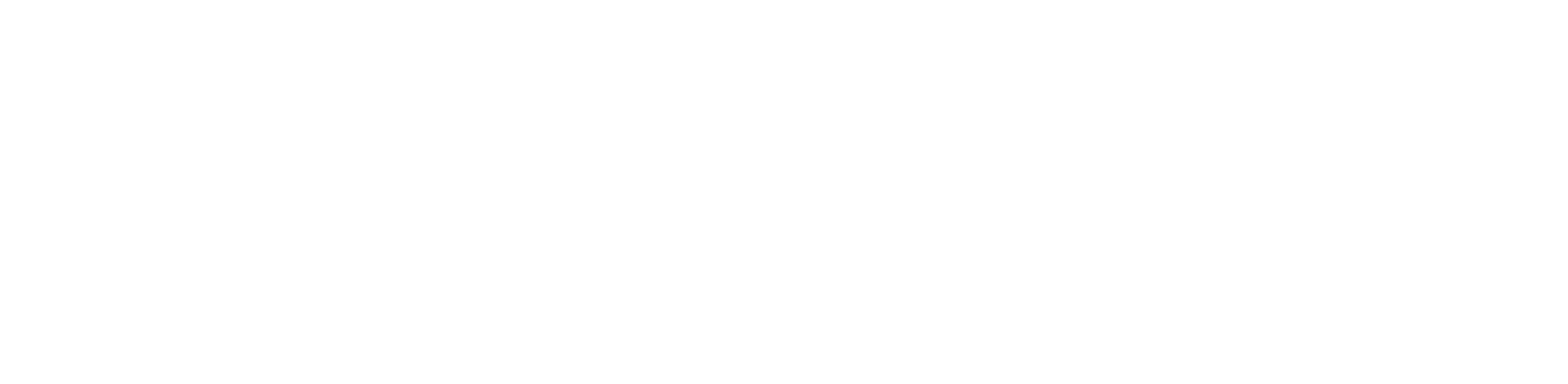
   // Constructor
   dynarray::dynarray(int s)
   {
       p = new int [s];
       if(!p) {
           cout << "Allocation error\n";
           exit(1);
       }

       size = s;
   }

   // Store an element.
   int &dynarray::put(int i)
   {
       if(i<0 || i>=size) {
           cout << "Bounds error!\n";
           exit(1);
       }

       return p[i];
   }

   // Get an element.
   int dynarray::get(int i)
   {
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
if(i<0 || i>size) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
}
return p[i];
}

int main()
{
    int i;

    dynarray ob1(10), ob2(10), ob3(100);

    ob1[3] = 10;
    i = ob1[3];
    cout << i << "\n";

    ob2 = ob1;

    i = ob2[3];
    cout << i << "\n";

    // generates an error
    ob1 = ob3; // arrays differ sizes
    return 0;
}
```

## **MASTERY SKILLS CHECK: Chapter 6**

1. // Overload << and >>.

```
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator<<(int i);
    coord operator>>(int i);
};
```

```
{  
    x = i; y = j; z = k;  
}  
three_d() { x=0; y=0; z=0; }  
void get(int &i, int &j, int &k)  
{  
    i = x; j = y; k = z;  
}  
three_d operator+(three_d ob2);  
three_d operator-(three_d ob2);  
three_d operator++();  
three_d operator--();  
};  
  
three_d three_d::operator+(three_d ob2)  
{  
    three_d temp;  
    temp.x = x + ob2.x;  
    temp.y = y + ob2.y;  
    temp.z = z + ob2.z;  
  
    return temp;  
}  
  
three_d three_d::operator-(three_d ob2)  
{  
    three_d temp;  
  
    temp.x = x - ob2.x;  
    temp.y = y - ob2.y;  
    temp.z = z - ob2.z;  
  
    return temp;  
}  
  
three_d three_d::operator++()  
{  
    x++;  
    y++;  
    z++;  
  
    return *this;  
}  
  
three_d three_d::operator--()
```

```
    if(o1>o2) cout << "o1 > o2\n";  
  
    if(o1<o2) cout << "o1 < o2\n";  
  
    return 0;  
}
```

## **R**EVIEW SKILLS CHECK: Chapter 7

1. No. Overloading an operator simply expands the data types upon which it can operate, but no preexisting operations are affected.
2. Yes. You cannot overload an operator relative to one of C++'s built-in types.
3. No, the precedence cannot be changed. No, the number of operands cannot be altered.

```
4. #include <iostream>  
using namespace std;  
  
class array {  
    int nums[10];  
public:  
    array();  
    void set(int n[10]);  
    void show();  
    array operator+(array ob2);  
    array operator-(array ob2);  
    int operator==(array ob2);  
};  
  
array::array()  
{  
    int i;  
    for(i=0; i<10; i++) nums[i] = 0;  
}  
  
void array::set(int *n)  
{  
    int i;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    friend array operator--(array &ob);
}

array::array()
{
    int i;

    for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << nums[i] << ' ';

    cout << "\n";
}

// Overload unary op using member function.
array array::operator++()
{
    int i;

    for(i=0; i<10; i++)
        nums[i]++;
}

return *this;
}

// Use a friend.
array operator--(array &ob)
{
    int i;

    for(i=0; i<10; i++)
```

2. The protected category is needed to allow a base class to keep certain members private while still allowing a derived class to have access to them.
3. No.

**7.3*****EXERCISES***

1. 

```
#include <iostream>
#include <cstring>
using namespace std;

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived : public mybase {
    int len;
public:
    myderived(char *s) : mybase(s) {
        len = strlen(s);
    }
    int getlen() { return len; }
    void show() { cout << get() << '\n'; }
};

int main()
{
    myderived ob("hello");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}
```
2. 

```
#include <iostream>
using namespace std;

// A base class for various types of vehicles.
class vehicle {
```

```
int num_wheels;
int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
        cout << "Wheels: " << num_wheels << '\n';
        cout << "Range: " << range << '\n';
    }
};

class car : public vehicle {
    int passengers;
public:
    car(int p, int w, int r) : vehicle(w, r)
    {
        passengers = p;
    }
    void show()
    {
        showv();
        cout << "Passengers: " << passengers << '\n';
    }
};

class truck : public vehicle {
    int loadlimit;
public:
    truck(int l, int w, int r) : vehicle(w, r)
    {
        loadlimit = l;
    }
    void show()
    {
        showv();
        cout << "loadlimit " << loadlimit << '\n';
    }
};

int main()
{
    car c(5, 4, 500);
```

```
    truck t(30000, 12, 1200);

    cout << "Car: \n";
    c.show();
    cout << "\nTruck:\n";
    t.show();

    return 0;
}
```

**7.4*****EXERCISES***

## 1. Constructing A

Constructing B

Constructing C

Destructuring C

Destructuring B

Destructuring A

## 2. #include &lt;iostream&gt;

using namespace std;

```
class A {
    int i;
public:
    A(int a) { i = a; }
};
```

```
class B {
    int j;
public:
    B(int a) { j = a; }
};
```

```
class C : public A, public B {
    int k;
public:
    C(int c, int b, int a) : A(a), B(b) {
        k = c;
    }
};
```

**7.5****EXERCISES**

2. A virtual base class is needed when a derived class inherits two (or more) classes, both of which are derived from the same base class. Without virtual base classes, two (or more) copies of the common base class would exist in the final derived class. However, if the original base is virtual, only one copy is present in the final derived class.

**MASTERY SKILLS CHECK: Chapter 7**

```
1. #include <iostream>
using namespace std;

class building {
protected:
    int floors;
    int rooms;
    double footage;
};

class house : public building {
    int bedrooms;
    int bathrooms;
public:
    house(int f, int r, double ft, int br, int bth) {
        floors = f;    rooms = r;    footage = ft;
        bedrooms = br; bathrooms = bth;
    }
    void show() {
        cout << "floors: " << floors << '\n';
        cout << "rooms: " << rooms << '\n';
        cout << "square footage: " << footage << '\n';
        cout << "bedrooms: " << bedrooms << '\n';
        cout << "bathrooms: " << bathrooms << '\n';
    }
};

class office : public building {
    int phones;
```

```
    int extinguishers;
public:
    office(int f, int r, double ft, int p, int ext) {
        floors = f; rooms = r; footage = ft;
        phones = p; extinguishers = ext;
    }
    void show() {
        cout << "floors: " << floors << '\n';
        cout << "rooms: " << rooms << '\n';
        cout << "square footage: " << footage << '\n';
        cout << "Telephones: " << phones << '\n';
        cout << "fire extinguishers: ";
        cout << extinguishers << '\n';
    }
};

int main()
{
    house h_ob(2, 12, 5000, 6, 4);
    office o_ob(4, 25, 12000, 30, 8);

    cout << "House: \n";
    h_ob.show();

    cout << "\nOffice: \n";
    o_ob.show();

    return 0;
}
```

2. When a base class is inherited as public, the public members of the base become public members of the derived class, and the base's private members remain private to the base. If the base is inherited as private, all members of the base become private members of the derived class.
3. Members declared as **protected** are private to the base class but can be inherited (and accessed) by any derived class. When used as an inheritance access specifier, **protected** causes all public and protected members of the base class to become protected members of the derived class.
4. Constructors are called in order of derivation. Destructors are called in **reverse** order.

```
cout.setf(ios::showpos);

cout << -10 << ' ' << 10 << '\n';

return 0;
}

2. #include <iostream>
using namespace std;

int main()
{
    cout.setf(ios::showpoint | ios::uppercase | 
               ios::scientific);

    cout << 100.0;

    return 0;
}

3. #include <iostream>
using namespace std;

int main()
{
    ios::fmtflags f;

    f = cout.flags(); // store flags

    cout.unsetf(ios::dec);
    cout.setf(ios::showbase | ios::hex);
    cout << 100 << '\n';

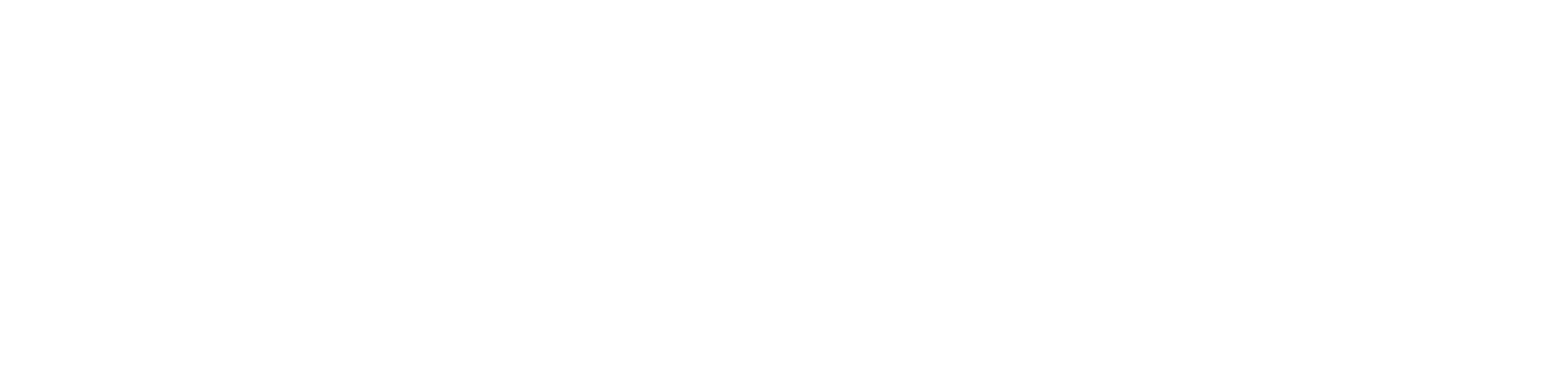
    cout.flags(f); // reset flags

    return 0;
}
```

## 8.3

**EXERCISES**

```
1 // Create a table of log10 and log from 2 through 100.
#include <iostream>
#include <cmath>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    stream << ob.p;

    return stream;
}

istream &operator>>(istream &stream, strtype &ob)
{
    char temp[255];

    stream >> temp;

    if(strlen(temp) >= ob.len) {
        delete [] ob.p;
        ob.len = strlen(temp)+1;
        ob.p = new char [ob.len];
        if(!ob.p) {
            cout << "Allocation error\n";
            exit(1);
        }
    }
    strcpy(ob.p, temp);

    return stream;
}

int main()
{
    strtype s1("This is a test."), s2("I like C++.");

    cout << s1;
    cout << '\n' << s2;

    cout << '\nEnter a string: ';
    cin >> s1;
    cout << s1;

    return 0;
}

2. #include <iostream>
using namespace std;

class factor {
    int num; // number
    int lfact; // lowest factor
public:
```

```
factor(int i);
friend ostream &operator<<(ostream &stream, factor ob);
friend istream &operator>>(istream &stream, factor &ob);
};

factor::factor(int i)
{
    int n;

    num = i;

    for(n=2; n < (i/2); n++)
        if(!(i%n)) break;

    if(n<(i/2)) lfact = n;
    else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
    stream >> ob.num;

    int n;

    for(n=2; n < (ob.num/2); n++)
        if(!(ob.num%n)) break;
    if(n<(ob.num/2)) ob.lfact = n;
    else ob.lfact = 1;

    return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
    stream << ob.lfact << " is lowest factor of ";
    stream << ob.num << '\n';

    return stream;
}

int main()
{
    factor o(32);
```

```
    tos--;
    return stck[tos];
}

ostream &operator<<(ostream &stream, stack ob)
{
    char ch;

    while(ch=ob.pop()) stream << ch;
    stream << endl;

    return stream;
}

int main()
{
    stack s;

    s.push('a');
    s.push('b');
    s.push('c');

    cout << s;
    cout << s;

    return 0;
}

2. #include <iostream>
#include <ctime>
using namespace std;

class watch {
    time_t t;
public:
    watch() { t = time(NULL); }
    friend ostream &operator<<(ostream &stream, watch ob);
};

ostream &operator<<(ostream &stream, watch ob)
{
    struct tm *localt;

    localt = localtime(&ob.t);
    stream << asctime(localt) << endl;
}
```

```
        return stream;
    }

    int main()
    {
        watch w;

        cout << w;

        return 0;
    }

3. #include <iostream>
using namespace std;

class ft_to_inches {
    double feet;
    double inches;
public:
    void set(double f) {
        feet = f;
        inches = f * 12;
    }
    friend istream &operator>>(istream &stream,
                                ft_to_inches &ob);
    friend ostream &operator<<(ostream &stream,
                                ft_to_inches ob);
};

istream &operator>>(istream &stream, ft_to_inches &ob)
{
    double f;

    cout << "Enter feet: ";
    stream >> f;
    ob.set(f);

    return stream;
}

ostream &operator<<(ostream &stream, ft_to_inches ob)
{
    stream << ob.feet << " feet is " << ob.inches;
    stream << " inches\n";
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
char name[80];
double balance;
public:
    account(int c, char *n, double b)
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }
    friend ostream &operator<<(ostream &stream, account ob);
};

ostream &operator<<(ostream &stream, account ob)
{
    stream << ob.custnum << ' ';
    stream << ob.name << ' ' << ob.balance;
    stream << '\n';

    return stream;
}

int main()
{
    account Rex(1011, "Ralph Rex", 12323.34);
    ofstream out("accounts", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    out << Rex;

    out.close();

    return 0;
}
```

## 9.4

**EXERCISES**

1. // Use get() to read a string that contains spaces.  
#include <iostream>  
#include <fstream>

```
int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "usage: SWAP <filename>\n";
        return 1;
    }

    // open file for input/output
    fstream io(argv[1], ios::in | ios::out | ios::binary);

    if(!io) {
        cout << "Cannot open file.\n";
        return 1;
    }

    char ch1, ch2;
    long i;

    for(i=0 ;!io.eof(); i+=2) {
        io.seekg(i, ios::beg);
        if(!io.good()) return 1;
        io.get(ch1);
        if(io.eof()) continue;
        io.get(ch2);
        if(!io.good()) return 1;
        if(io.eof()) continue;
        io.seekg(i, ios::beg);
        if(!io.good()) return 1;
        io.put(ch2);
        if(!io.good()) return 1;
        io.put(ch1);
        if(!io.good()) return 1;
    }

    io.close();
    if(!io.good()) return 1;

    return 0;
}
```

```
4. // Count letters.

#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: COUNT <source>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    // init alpha[]
    int i;
    for(i=0; i<26; i++) alpha[i] = 0;

    while(!in.eof()) {
        ch = in.get();
        if(isalpha(ch)) { // if letter found, count it
            ch = toupper(ch); // normalize
            alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, etc.
        }
    };

    // display count
    for(i=0; i<26; i++) {
        cout << (char) ('A'+ i) << ":" << alpha[i] << '\n';
    }

    in.close();

    return 0;
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        break;
    }
    p2 = p;
    p = p->next;
}
if(!p) { // goes on end
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
}
if(!head) // is first element
    head = item;
}

int sorted::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // remove from start of list
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

int main()
{
    list *p;

    // demonstrate queue
    queue q_obj;
    p = &q_obj; // point to queue

    p->store(1);
    p->store(2);
    p->store(3);
```

```
cout << "Queue: ";
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();

cout << '\n';

// demonstrate stack
stack s_ob;
p = &s_ob; // point to stack

p->store(1);
p->store(2);
p->store(3);

cout << "Stack: ";
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();

cout << '\n';

// demonstrate sorted list
sorted sorted_ob;
p = &sorted_ob;

p->store(4);
p->store(1);
p->store(3);
p->store(9);
p->store(5);

cout << "Sorted: ";
cout << p->retrieve();

cout << '\n';

return 0;
}
```

```
};

// Create a queue-type list.
class queue : public list {
public:
    void store(int i);
    int retrieve();
    queue operator+(int i) { store(i); return *this; }
    int operator--(int unused) { return retrieve(); }
};

void queue::store(int i)
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Allocation error.\n";
        exit(1);
    }
    item->num = i;

    // put on end of list
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
    if(!head) head = tail;
}

int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // remove from start of list
    i = head->num;
    p = head;
    head = head->next;
    delete p;
```

```
        return i;
    }

    // Create a stack-type list.
class stack : public list {
public:
    void store(int i);
    int retrieve();
    stack operator+(int i) { store(i); return *this; }
    int operator--(int unused) { return retrieve(); }
};

void stack::store(int i)
{
    list *item;

    item = new stack;
    if(!item) {
        cout << "Allocation error.\n";
        exit(1);
    }
    item->num = i;

    // put on front of list for stack-like operation
    if(head) item->next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // remove from start of list
    i = head->num;
    p = head;
    head = head->next;
    delete p;
```

**REVIEW SKILLS CHECK: Chapter 11**

1. A virtual function is a function that is declared as **virtual** by a base class and then overridden by a derived class.
2. A pure virtual function is one that has no body defined within the base class. This means that the function must be overridden by a derived class. A base class that contains at least one pure virtual function is called an abstract class.
3. The missing words are "virtual" and "base".
4. If a derived class does not override a non-pure virtual function, the derived class will use the base class's version of the virtual function.
5. The main advantage of run-time polymorphism is flexibility. The main disadvantage is loss of execution speed.

**11.1****EXERCISES**

```
2. #include <iostream>
using namespace std;

template <class X> X min(X a, X b)
{
    if(a<=b) return a;
    else return b;
}

int main()
{
    cout << min(12.2, 2.0);
    cout << endl;
    cout << min(3, 4);
    cout << endl;
    cout << min('c', 'a');
    return 0;
}

3. #include <iostream>
#include <cstring>
using namespace std;

template <class X> int find(X object, X *list, int size)
```

```
{  
    int i;  
  
    for(i=0; i<size; i++)  
        if(object == list[i]) return i;  
    return -1;  
}  
  
int main()  
{  
    int a[] = {1, 2, 3, 4};  
    char *c = "this is a test";  
    double d[] = {1.1, 2.2, 3.3};  
  
    cout << find(3, a, 4);  
    cout << endl;  
    cout << find('a', c, (int) strlen(c));  
    cout << endl;  
    cout << find(0.0, d, 3);  
  
    return 0;  
}
```

4. Generic functions are valuable because they allow you to define a general algorithm that can be applied to various types of data. (That is, specific versions of the algorithm need not be explicitly created by you.) Generic functions further help implement the concept of "one interface, multiple methods," which is a common theme in C++ programming.

**11.2****EXERCISES**

2. // Create a generic queue.
- ```
#include <iostream>  
using namespace std;  
  
#define SIZE 100  
  
template <class Qtype> class q_type {  
    Qtype queue[SIZE]; // holds the queue  
    int head, tail; // indices of head and tail  
public:  
    q_type() { head = tail = 0; }
```

```
void q(Qtype num); // store
Qtype deq(); // retrieve
};

// Put value on queue.
template <class Qtype> void q_type<Qtype>::q(Qtype num)
{
    if(tail+1==head || (tail+1==SIZE && !head)) {
        cout << "Queue is full.\n";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // cycle around
    queue[tail] = num;
}

// Remove value from queue.
template <class Qtype> Qtype q_type<Qtype>::deq()
{
    if(head == tail) {
        cout << "Queue is empty.\n";
        return 0; // or some other error indicator
    }
    head++;
    if(head==SIZE) head = 0; // cycle around
    return queue[head];
}

int main()
{
    q_type<int> q1;
    q_type<char> q2;
    int i;

    for(i=1; i<=10; i++) {
        q1.q(i);
        q2.q(i-1+'A');
    }

    for(i=1; i<=10; i++) {
        cout << "Dequeue 1: " << q1.deq() << "\n";
        cout << "Dequeue 2: " << q2.deq() << "\n";
    }
}
```

```
        return 0;
    }

3. #include <iostream>
using namespace std;

template <class X> class input {
    X data;
public:
    input(char *s, X min, X max);
    // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
{
    do {
        cout << s << ": ";
        cin >> data;
    } while( data < min || data > max);
}

int main()
{
    input<int> i("enter int", 0, 10);
    input<char> c("enter char", 'A', 'Z');

    return 0;
}
```

**11.3*****EXERCISES***

2. The **throw** is called before execution passes through a **try** block.
3. A character exception is thrown, but the **catch** statement will handle only a character pointer. (That is, there is no corresponding **catch** statement to handle the character exception.)
4. If an exception is thrown for which there is no corresponding **catch, terminate( )** is called and abnormal program termination might occur.

```
}

try {
    p = new int;
} catch(bad_alloc ba) {
    cout << "Allocation error.\n";
    // ...
}
```

## **MASTERY SKILLS CHECK: Chapter 11**

```
1. #include <iostream>
#include <cstring>
using namespace std;

// A generic mode-finding function.
template <class X> X mode(X *data, int size)
{
    register int t, w;
    X md, oldmd;
    int count, oldcount;

    oldmd = 0;
    oldcount = 0;
    for(t=0; t<size; t++) {
        md = data[t];
        count = 1;
        for(w = t+1; w < size; w++)
            if(md==data[w]) count++;
        if(count > oldcount) {
            oldmd = md;
            oldcount = count;
        }
    }
    return oldmd;
}

int main()
{
    int i[] = { 1, 2, 3, 4, 2, 3, 2, 2, 1, 5};
    char *p = "this is a test";
```

```
    }

}

int main()
{
    int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
    double d[] = {1.2, 5.5, 2.2, 3.3};
    int j;

    bubble(i, 10); // sort ints
    bubble(d, 4); // sort doubles

    for(j=0; j<10; j++) cout << i[j] << ' ';
    cout << endl;

    for(j=0; j<4; j++) cout << d[j] << ' ';
    cout << endl;

    return 0;
}

4. /* This function demonstrates a generic stack that
   holds two values. */
#include <iostream>
using namespace std;

#define SIZE 10

// Create a generic stack class
template <class StackType> class stack {
    StackType stck[SIZE][2]; // holds the stack
    int tos; // index of top of stack

public:
    void init() { tos = 0; }
    void push(StackType ob, StackType ob2);
    StackType pop(StackType &ob2);
};

// Push objects.
template <class StackType>
void stack<StackType>::push(StackType ob, StackType ob2)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
    }
}
```

```
        return;
    }
    stck[tos][0] = ob;
    stck[tos][1] = ob2;
    tos++;
}

// Pop objects.
template <class StackType>
StackType stack<StackType>::pop(StackType &ob2)
{
    if(tos==0) {
        cout << "Stack is empty.\n";
        return 0; // return null on empty stack
    }
    tos--;
    ob2 = stck[tos][1];
    return stck[tos][0];
}

int main()
{
    // Demonstrate character stacks.
    stack<char> s1, s2; // create two stacks
    int i;
    char ch;

    // initialize the stacks
    s1.init();
    s2.init();

    s1.push('a', 'b');
    s2.push('x', 'z');
    s1.push('b', 'd');
    s2.push('y', 'e');
    s1.push('c', 'a');
    s2.push('z', 'x');

    for(i=0; i<3; i++) {
        cout << "Pop s1: " << s1.pop(ch);
        cout << ' ' << ch << "\n";
    }
    for(i=0; i<3; i++) {
        cout << "Pop s2: " << s2.pop(ch);
        cout << ' ' << ch << "\n";
    }
}
```

```
}

// demonstrate double stacks
stack<double> ds1, ds2; // create two stacks
double d;

// initialize the stacks
ds1.init();
ds2.init();

ds1.push(1.1, 2.0);
ds2.push(2.2, 3.0);
ds1.push(3.3, 4.0);
ds2.push(4.4, 5.0);
ds1.push(5.5, 6.0);
ds2.push(6.6, 7.0);

for(i=0; i<3; i++) {
    cout << "Pop ds1: " << ds1.pop(d);
    cout << ' ' << d << "\n";
}

for(i=0; i<3; i++) {
    cout << "Pop ds2: " << ds2.pop(d);
    cout << ' ' << d << "\n";
}

return 0;
}
```

5. The general forms of **try**, **catch**, and **throw** are shown here:

```
try {
    // try block
    throw exp;
}
catch(type arg) {
    // ...
}
```

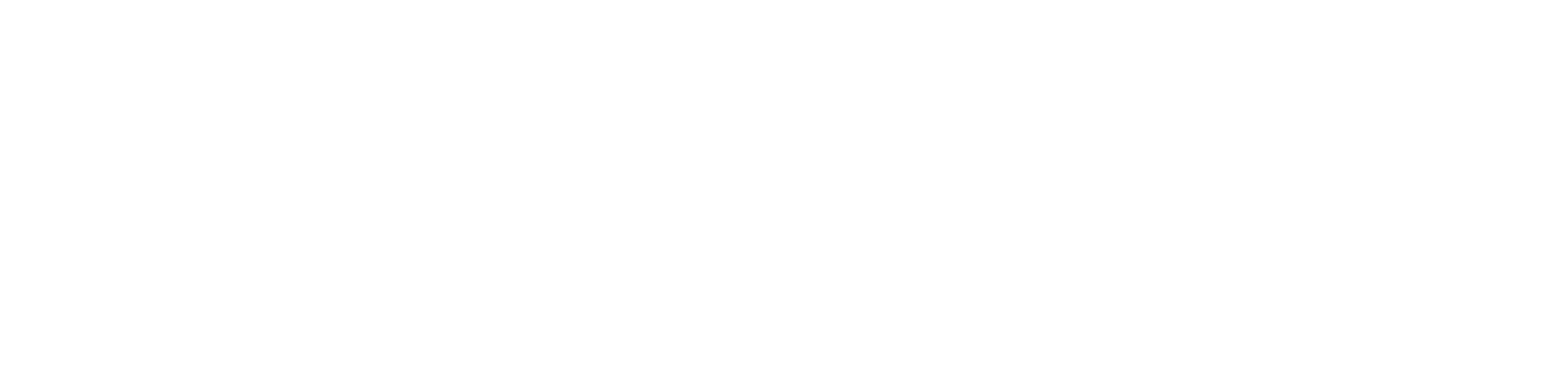
6. /\* This function demonstrates a generic stack  
that includes exception handling. \*/  
#include <iostream>



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
try {
    switch(rand() % 4) {
        case 0:
            return new Line;
        case 1:
            return new Rectangle;
        case 2:
            return new Triangle;
        case 3:
            return new NullShape;
    }
} catch (bad_alloc ba) {
    return NULL;
}
return NULL;
}
```

2. Here is **generator( )** recoded to use **new(nothrow)**

```
// Use new(nothrow)
Shape *generator()
{
    Shape *temp;

    switch(rand() % 4) {
        case 0:
            temp = new(nothrow) Line;
            break;
        case 1:
            temp = new(nothrow) Rectangle;
            break;
        case 2:
            temp = new(nothrow) Triangle;
            break;
        case 3:
            temp = new(nothrow) NullShape;
    }

    if(temp) return temp;
    else return NULL;
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    /* display projects */
    while(p != proj.end()) {
        p->report();
        p++;
    }

    return 0;
}
```

**14.5****EXERCISES**

2. // A map of names and phone numbers.

```
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
    char str[20];
public:
    name() { strcpy(str, ""); }
    name(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

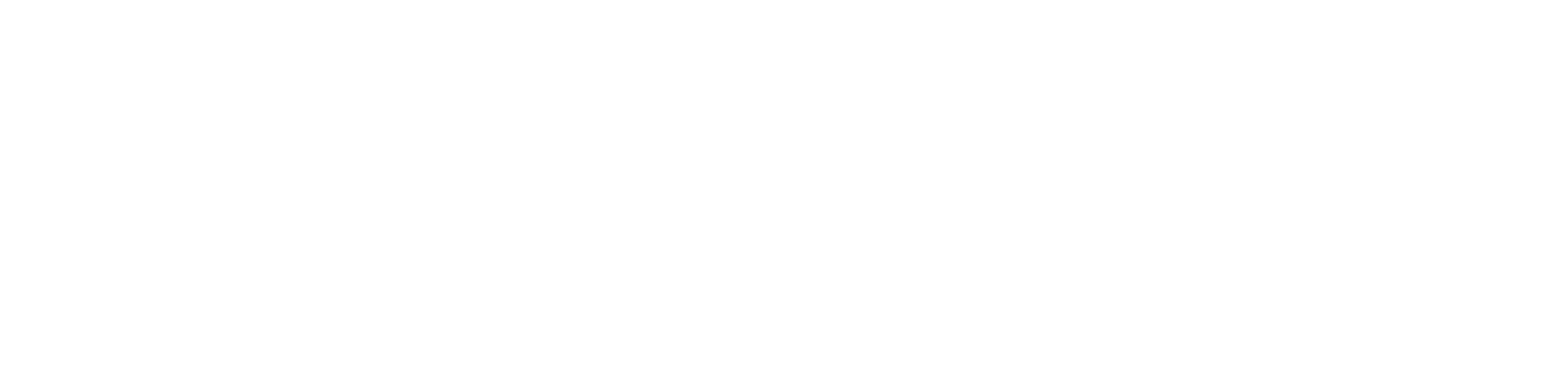
// must define less than relative to name objects
bool operator<(name a, name b)
{
    return strcmp(a.get(), b.get()) < 0;
}

class phonenum {
    char str[20];
public:
    phonenum() { strcpy(str, ""); }
    phonenum(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

int main()
{
    map<name, phonenum> m;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**F**unction(s)

conversion, 446-449  
friend, 107-113  
generated, 375  
generic. *See* Generic functions  
in assignment statements, using, 149-150  
in-line. *See* In-line functions  
member. *See* Member functions  
objects, 478  
operator. *See* Operator functions  
parameterless, 28-29  
passing objects to, 96-101  
passing references to, 140-145  
pointers to overloaded, 189-191  
predicate, 477-478  
prototyping, 29, 30-31  
return value and, 29  
returning objects from, 102-106  
returning references from, 149-153  
virtual. *See* Virtual functions

**G**

gcount( ) function, 321  
Generated function, 375  
Generic class(es), 274, 372, 380-386  
declaration, general form of, 380  
with multiple generic data types, 385-386  
Generic functions, 372, 373-379, 380-386  
explicitly overloading, 378-379  
general form for, 373  
with multiple generic types, 376-377  
versus overloaded functions, 377, 379  
get( ) function, 321, 322, 323, 327, 328  
Get pointer, 332, 333-334  
getc( ) function, 327  
getline( ) function, 327-328, 329  
good( ) function, 335, 337  
goodbit flag, 335

**H**

Headers, 8-11, 444  
hex  
format flag, 275, 276, 279

I/O manipulator, 288  
Hierarchical classification, 6, 7, 59

**I**

ifstream class, 275, 313  
#include statement, 9-10, 11  
Increment operator (+ +), overloading for  
postfix and prefix, 210-211, 217-218  
Inheritance, 4, 6, 7, 59-65  
and class access control, 234-243  
and constructors and destructors, 244-249  
and friend functions, 109  
general form for class, 234  
multiple, 252-257  
and virtual base classes, 259-261  
Initialization of objects, 43-44, 46-47, 134  
copy constructors and, 168-171  
overloaded constructors and, 161-166

**In-line functions**, 75-79

automatic, 80-82  
to define constructors and destructors, 81-82  
versus parameterized macros, 77

**inline specifier**, 76

Input operator (>>), 13-14, 299, 338  
insert( ) function, 482, 484, 487-488, 491, 492,  
523, 527-528

Inserters (inserter functions), creating,  
292-297

Insertion operator (<<). *See* Output operator

int, default to, 29

**internal**

format flag, 275, 276  
I/O manipulator, 288

**I/O**

array-based, 466-470  
binary. *See* Binary I/O  
console, 13-18, 270, 338  
customized, and files, 338-340  
file. *See* File I/O  
formatted, 275-291  
inserters and extractors, 292-302  
library, 270



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

static\_cast, 429, 431  
 std namespace, 11, 437, 442-444  
 stderr stream, 273  
 stdin stream, 273  
 stdio.h header file, 10  
 stdout stream, 13, 273  
 STL (Standard Template Library), 380, 474, 476-478  
 strcat( ) function, 522  
 strcpy( ) function, 449, 520, 522!  
 streambuf class, 274  
 Streams (I/O), 273-274, 313  
 streamsiz type, 284, 321  
 string class, 474, 519-525  
     operators defined for, 521-522  
 String handling in C++, 519-520, 525-527  
 <string> header, 521  
 string.h header file, 10  
 strstream class, 466, 467  
 <strstream> header, 466  
 struct keyword, 68-69, 71  
 Structured programming, 3  
 Structures in C++, 68-74

**I**

tellg( ) function, 332  
 tellp( ) function, 332  
 Template functions. *See Generic functions*  
 template statement, 373, 375-376, 380  
 Templates, 372  
 terminate( ) function, 388, 395  
 this pointer, 126-129  
     and friend operator functions, 213  
     and inserters, 292-293  
     and member operator functions, 199  
 throw, 386-388, 395  
     statement, general form for, 388  
     clause, general form for, 394-395  
 time\_t type, 165  
 transform( ) algorithm, 512, 516-518  
 True and false in C and C++, 30  
 true value, 30, 207

try statement, 386-388, 389-392, 398  
 Type conversions and ambiguity, automatic, 185-187  
 typeid operator, 409, 410-419, 422, 426  
 type\_info, 409, 413  
 <typeinfo> header, 409  
 typename keyword, 374, 376

**U**

Unary operators, overloading, 209-212  
 unexpected( ) function, 395, 398  
 Unions, 69-70, 72-75  
 unitbuf  
     format flag, 275, 276  
     I/O manipulator, 288  
 unsetf( ) function, 277, 289  
 uppercase  
     format flag, 275, 276, 279  
     I/O manipulator, 288  
 using statement, 438-439  
 <utility> header, 478

**V**

Variables  
 global, declaring, 532  
 global vs. static, 450  
 local, declaring, 29-30, 31  
 objects as, 5  
 public member, 25-26  
 vector class, 476, 479, 480-490  
     member functions, table of, 482-483  
     using iterator to access, 486-487  
 Virtual base class, 259-261  
 Virtual functions, 349-361  
     hierarchical nature of, 352-354, 361  
     and late binding, 363  
     overriding, 351-357  
     and polymorphism, 346, 349-350, 362-368  
     pure, 358-360  
     responding to run-time random events  
         with, 354-355, 366-368



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# TEACH YOURSELF C++ Third Edition

## The Most Effective Way to Learn C++

Programming luminary Herb Schildt is back with a fresh, new edition of his best-selling C++ tutorial! In this third edition of *Teach Yourself C++*, Herb's proven plan for success has been thoroughly updated, expanded, and enhanced. Whether you're a C programmer moving up to C++ or an experienced C++ pro looking for coverage of this language's hottest new features, there truly is no better way to learn.

Written with uncompromising clarity and the attention to detail that has made Herb Schildt famous, *Teach Yourself C++ Third Edition* begins with the fundamentals, covers all the essentials and concludes with a look at some of C++'s most advanced features. Along the way are plenty of practical examples, self-evaluation skill checks, and exercises—with answers at the back of the book so you can easily check your progress.

Several new elements have been added to C++, such as namespaces, runtime type ID, the new casting operators, and the Standard Template Library. Herb covers them all and because Herb teaches Standard C++, you can be assured that what you learn today will still apply tomorrow.

### Inside you will

- ✓ Learn the principles of Object Oriented Programming (OOP)
- ✓ Understand the structure of a C++ program
- ✓ Explore classes, the building blocks of C++
- ✓ Work with constructors and destructors
- ✓ Understand function- and operator- overloading
- ✓ Learn about inheritance
- ✓ Investigate the C++ I/O System
- ✓ Discover the power of virtual functions
- ✓ Work with templates and exception handling
- ✓ Understand Runtime Type ID and the Casting Operators
- ✓ Utilize namespaces
- ✓ Explore the Standard Template Library (STL)

Clearly structured and guaranteed to bring you successful results, *Teach Yourself C++ Third Edition* is the hands-down, easiest way to master one of today's hottest programming languages.

*Works with all C/C++ Compilers*

*Covers Standard Template Library*

*The McGraw-Hill Companies*



**Tata McGraw-Hill  
Publishing Company Limited**  
**7 West Patel Nagar, New Delhi 110 008**

Visit our website at : [www.tatamcgrawhill.com](http://www.tatamcgrawhill.com)

ISBN-13: 978-0-07-463870-5  
ISBN-10: 0-07-463870-X



9 780074 638705

Copyrighted material